An alternative implementation schema for ASSIST `parmod`

M. Danelutto, C. Migliore & C. Pantaleo
Dept. Computer Science
University of Pisa
marcod@di.unipi.it
{c.migliore,pantal}@libero.it

Abstract

ASSIST is a structured parallel programming environment targeting networks/clusters of workstations and grids. It introduced the parmod *parallel construct, supporting a variety of parallelism exploitation patterns, including classical ones. The original implementation of* parmod *relies on static assignment of parallel activities to the processing elements at hand. In this work we discuss an alternative implementation of the* parmod *construct that implements completely dynamic assignment of parallel activities to the processing elements. We show that the new implementation introduces very limited overhead in case of regular computations, whereas it performs much better than the original one in case of irregular applications. The whole implementation of* parmod *is available as a C++/MPI library.*

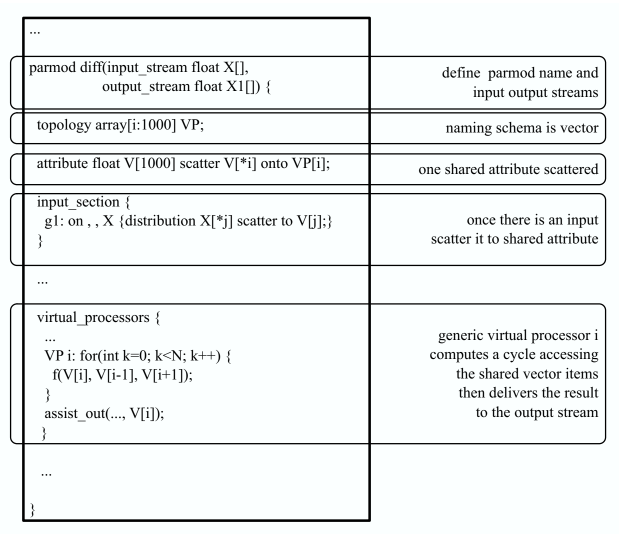
Keywords*: algorithmic skeletons, data flow, data parallelism, stream parallelism, irregular computation, auomatic load balancing.*

1 Introduction

Structured parallel programming models usually provide the programmer with a set of language constructs/library calls that take full care of implementing common parallelism exploitation patterns. In the algorithmic skeletons framework, such constructs are called *skeletons* and can be provided either as library calls [13, 9] or as language constructs [5, 6] In the design pattern context, such constructs are just provided as *parallel design patterns* [14, 15]. There are also several coordination languages that provide these constructs as coordination schemas/keywords [1]. The main and more common parallelism exploitation patterns modeled by the structured parallel programming environment constructs include *task farm* (aka embarrassingly parallel computations), *pipeline*, *map* (independent forall), and *divide&conquer*. Just to have an idea, all what a programmer has to do to write an embarrassingly parallel application processing a stream of images with some kind of structured parallel programming environment is to choose the task farm construct and tell the compiler/library implementing such construct which is the computation (that is tell it the pointer to the code) actually performing the computation on the single input image. Independently of the framework, all the structured parallel programming models suffer from the same very basic problem (see also the discussion in Cole's "manifesto" in [8]): the set of skeletons/parallel design patterns/coordination schemas are usually fixed (with the only exception of the work discussed in [14]). This is to optimize implementation, of course. As the programmers must just pick up a construct and provide it functional parameters to implement a parallel application, efficient implementation of these constructs is very complex because it demands efficient implementation of parallel activities, communication and synchronization setup and management. It is even more complex if the system provides the ability to nest constructs, as it happens in P3L [5]. On the other hand, structured parallel programming systems have been assessed to be much more usable than classical message passing parallel programming tools [18].

Let's take into account how the task farm computation processing images mentioned above could be implemented. A set of available processing elements has to be found, image processing code has to be staged to the processing elements, then images must be scheduled to the available processing elements to be actually computed, taking into account that load must be balanced, the communications cost, etc.

However, most programmers of parallel implementation eventually feel that they would like to slightly change the constructs provided to behave more accordingly to their specific needs and this is of course not possible without releasing the constrain that implementation is completely opaque to the programmer. Again, as an example, the stream image processing application may need to count some feature of the processed images and the counter re-



```
...
parmod diff(input_stream float X[],
             output_stream float X1[]) {
 topology array[i:1000] VP;
 attribute float V[1000] scatter V[*i] onto VP[i];
 input_section {
  g1: on , , X {distribution X[*j] scatter to V[j];}
 }
 ...
 virtual_processors {
  ...
  VP i: for(int k=0; k<N; k++) {
   f(V[i], V[i-1], V[i+1]);
  }
  assist_out(..., V[i]);
 }
  ...
}
```

define parmod name and input output streams
naming schema is vector
one shared attribute scattered
once there is an input scatter it to shared attribute
generic virtual processor i computes a cycle accessing the shared vector items then delivers the result to the output stream

Figure 1. Sketch of ASSIST `parmod` **code**

quires global communication among the processing elements involved. Either this was somehow planned by the original implementation, or control has to be provided to the programmer over the whole implementation to program its own task farm variant.

Recently, our group come up with the definition of a new structured parallel programming environment, ASSIST [19] that tries to solve this problem introducing a generic parallel skeleton, the `parmod` that can be customized to behave like several different parallelism exploitation patterns.

The `parmod` construct requires (and allows) the programmer to specify more parameters with respect to classical structured parallel programming constructs, namely:

- a set of logically parallel activities, named *virtual processors*, along with a naming schema, i.e. a way to name these virtual processors, and the code to be executed by each virtual processor or by each partition of the virtual processors. As an example, a programmer can define a vector of virtual processors, with the even index ones computing a given function of the input data and the odd index ones computing another function

- the way input data are delivered to the virtual processors for processing. `parmod` processes streams of input data (tasks) to produce stream of output data (results). A parmod can have multiple input streams and the programmer must specify the (possibly nondeterministic) control specifying how input data are delivered to the virtual processors. Input data can be scattered or broadcasted to the virtual processors. They can also be sent to just one virtual processor or in multicast to virtual processors partitions.

- the way data is collected from the virtual processors to be delivered onto the output streams. Again, one data per virtual processor can be collected to build a single data item to be placed on one of the output streams or data coming from any of the virtual processors can be simply delivered to the output streams.

- the way a shared state is managed by the virtual processors. State variables can be shared by the virtual processors according to the owner computes rule. Shared data structured can be scattered, broadcasted or multicasted to the `parmod` virtual processors.

As all these items can be independently specified; the resulting `parmod` set is quite large and includes, as sub cases both classical parallelism exploitation patterns and more specific ones. As an example, task farm can be implemented by specifying no naming for the virtual processors (anonymous workers), delivering each data item appearing onto an input stream to any idle worker (`on demand` distribution policy) and delivering any data output computed by each virtual processor to the output stream. Instead, an independent forall computation computing items of a shared vector can be implemented by specifying a `vector` naming for the virtual processors, by scattering the shared vector onto the virtual processors, then scattering each input data to the virtual processors and eventually gathering the final shared vector values to deliver the shared vector to the output stream. We do not want to enter in the details of the ASSIST `parmod` technicalities in this work (the interested reader can refer to [19] or to the ASSIST web site `http://www.di.unipi.it/Assist.html`). Rather, we want to concentrate onto the implementation of `parmod`. Therefore, in Section 2 we discuss the classical implementation of `parmod`. Then in Section 3 we will describe a new implementation schema and in Section 4 we eventually compare the results achieved using both implementations on a class of typical problems.

2 Standard `parmod` implementation

Virtual processor execution is always implemented in the same way, independently of the different, naming, distribution and collection policies and shared state variables specified by the programmer: each input data set for virtual processor is computed (that is remote data is retrieved, if any) then the virtual processor is scheduled for the execution on one of the available processing elements. The current implementation of ASSIST manages to figure out at compile time a sort of static allocation of virtual processors to actual processing elements. This static allocation tries to minimize the amount of data that need to be communicated to build each virtual processor input data set. As an example, consider a `parmod` such as the one sketched in Figure 1. The `parmod` computes a number of iterations, substituting at each iteration the current value of a state vector with the value obtained by computing the average of the adjacent vector items. Such computation is started every time a new data item appears on the input stream. Despite the vector topology (i.e. naming schema) this is a very common situation found in most code computing differential equations solutions.

In this case, virtual processors are partitioned across the available processing elements in a *block* way. Therefore, at each iteration, only the boundary values of the shared vector must be exchanged/communicated between different processing elements in order to have all the 1000 virtual processors ready to compute their `f` on plain local data. This kind of implementation has pros and cons, of course. The pros are mainly related to the possibility to compute at compile time the amount of data that has to be exchanged/moved across processing elements at run time. This is also guaranteed by the impossibility to use arbitrary index expression in the virtual process calls. The cons are mainly related to the impossibility to determine at run time *irregular* partitions of virtual processors to processing elements to match the load imbalance implicit in the kind of computation at hand. As an example, let us suppose that the first half of the vector requires a light computation while `f` takes much more to be computed in the second half of the vector. In this case, current implementation of ASSIST places anyway an equal number of virtual processors on each one of the processing elements available. Then, a parmod *manager* tries to redistribute virtual processors when load imbalances are observed [4].

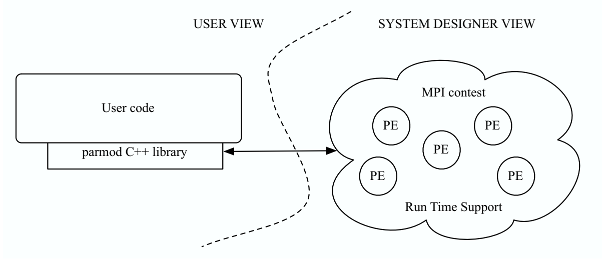


Figure 2. Dynamic implementation logical view

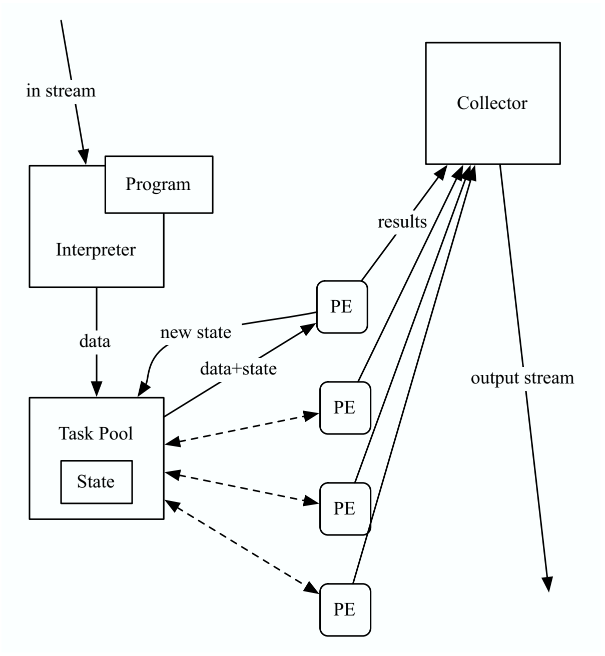


Figure 3. Internal organization of the dynamic `parmod` library

3 Dynamic implementation

In this work we propose a completely different implementation of `parmod`. Basically, it is derived from macro data flow skeleton implementation as proposed in [10, 11]. In this perspective, virtual processors are considered macro data flow instructions, as the computation associated to each one of the virtual processors can be started as soon as all the input parameters are available, both the ones coming from the input streams and the ones retrieved from shared state variables. Therefore, according to our dynamic implementation schema, virtual processors represent the data input to a macro data flow interpreter. In turn, the macro data flow interpreter is responsible of managing data flow across the virtual processors. E.g. to route the data tokens produced by one virtual processor to the proper place: output streams, shared state variables or next iteration virtual processor computations. As the original ASSIST implementation of `parmod` actually relies on a centralized management of input and output streams (that is input data comes to a unique, centralized process and output data are delivered by a unique, centralized process) we designed a centralized `parmod` interpreter structure, such as the one of Figure 3. The interpreter module is in charge of accepting input data items from the input streams and arranging them in such a way they can be used by the proper virtual processors. The result is either the addition of data items in existing virtual processors already stored in the task pool module, or the insertion of new instances of virtual processors in the same task module. The task module is the one responsible of delivering tasks to the remote executors (alias macro data flow interpreters). It is multithreaded, with one thread taking care of the management of a single remote executor. Each one of these threads searches for a fireable macro data flow instruction, that is for a virtual processor with all its input data already available. When such a virtual processor is found, it is delivered for execution to the remote executor associated to the thread. Eventually, the remote executor will send back a result. The result is either a state update or a mark stating that a final result has been computed and delivered to the collector, to be placed onto the proper output stream. In case of state updates, the thread will take care of routing it to the proper place, possibly making fireable another virtual processor instance. Then the loop is restarted, looking for a new fireable instruction to be delivered to the remote executor.

This happens much in the same way traditional data flow engines look for fireable data flow instructions in the matching unit and the deliver them for execution to a machine taken from a pool of machines. This dynamic `parmod` implementation is realized via a C++ library exploiting MPI. The goal is to provide the user with a completely transparent library as shown in Figure 2.

As an example, consider the ASSIST `parmod` of Figure 1. In this case, because of the virtual processor definition, the task pool will hold initially virtual processors with just an empty data item relative to the scattered shared state variable. As soon as the interpreter delivers the data coming from the input data stream, all the virtual processors become fireable. The threads blocked looking for fireable instructions unblock and start delivering tasks to be computed to the remote executor nodes. Those nodes actually execute a loop. Each virtual processor reads, during each loop iteration, the values of the shared variable hold by other virtual processors which are provided by the task pool, and eventually delivers new value for the owned portion of the shared variables to the task pool. The shared data values are thus available through the task pool module that manages to properly synchronize the virtual processor loop execution.

The original `parmod` is implemented as a construct in the coordination language of ASSIST. Therefore programmers actually write code similar to the one of Figure 1 and then this code is compiled to *task code*, a sort of C++ parallel assembly code that runs on top of plain TCP/IP POSIX frameworks [2] as well as on top of the Globus grid middleware [7]. Instead, the dynamic implementation of `parmod` is actually a C++ library calling MPI. Therefore, programmer code looks like very different. As an example, to do the same job of the single line of ASSIST code:

```
topology array [i:10] VP;
```

that states that there are 10 virtual processors, whose names are `VP[0]` to `VP[9]`, programmers must write the following code:

```
VP = Pm->createVP();
Index i;
i.setRange(1,10);
VP->setDomain(i);
```

Summarizing, the expressive power of our dynamic `parmod` library is definitely worst than the one of ASSIST. However it has to be taken into account that this library is thought as a substitute of the implementation of ASSIST `parmod` rather than as a library to be provided to the user, and therefore its expressive power, being used as a target of an automatic code generation, is ok. Also, as in the next Section we will compare the results achieved using standard ASSIST compiler (that is running on TCP/IP sockets) with results achieved running our dynamic `parmod` library (that is running on top of `mpich` that in turn runs on top of TCP/IP sockets on the machines used for the experiments) it must be taken into account that in our library there is a (thin) additional interpretation layer, the one involving MPI. Although using `mpich` the overhead involved is very low, this represents another significant difference with the current ASSIST implementation. When used to target workstation clusters/networks, in fact, the current ASSIST compiler generates code that directly uses TCP/IP sockets, without any additional layer on top of it.

The logically centralized data structure, as well as the logically unique list of ready virtual processors looks like actual bottlenecks preventing this schema from scaling beyond a small number of processing elements. In this paper we actually discuss results achieved using a centralized implementation but we already experimented the possibility to efficiently implement the logically centralized task pool in a distributed, scalable way [12].

4 Experiments

We performed several experiments to validate the dynamic implementation schema of `parmod`. We first measured scalability of our library. Then we measured performance (completion times, actually) of our library against the performance of ASSIST both in case where the static distribution of virtual processors to processing elements turns out to be good (no load imbalance) and in case it turns out to be definitely not good. Eventually we measured the impact of using state variables accessed from the different virtual processors.

In this work, we only show an excerpt of the overall set of results. In particular we show "extreme case" results, in a sense, i.e. results that show how the dynamic `parmod` implementation performs under extreme conditions. This is to

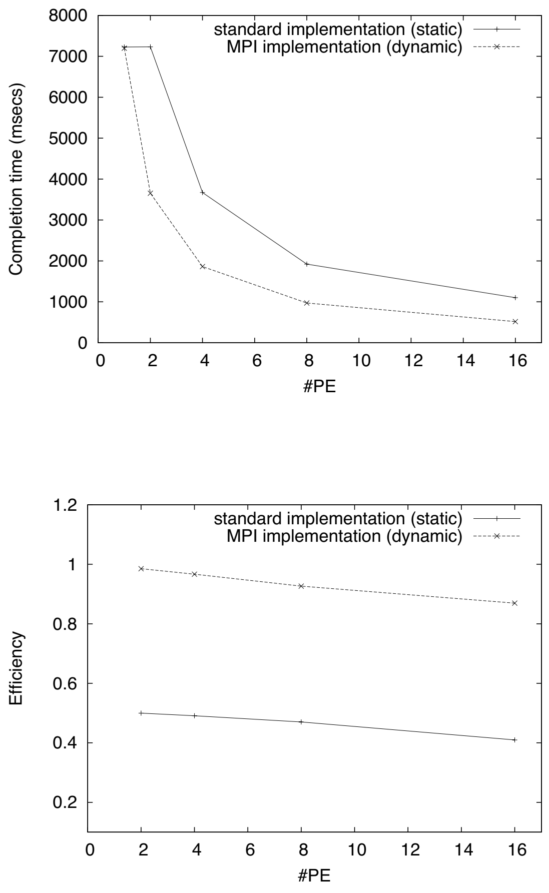


Figure 4. Dynamic (MPI `parmod`) vs. static (ASSIST) implementation of `parmod`: unbalanced computation case: completion times (upper) and efficiency (lower)

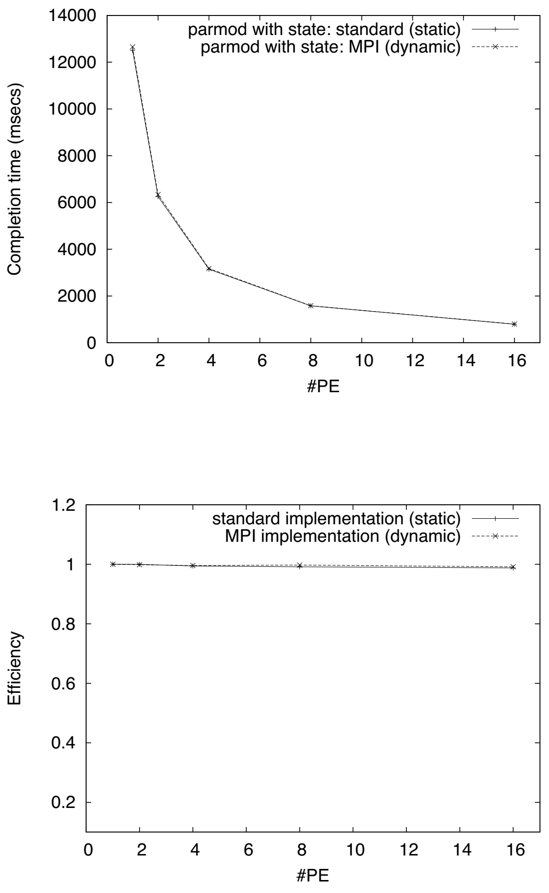


Figure 5. Dynamic (MPI `parmod`) vs. static (ASSIST) implementation of `parmod`: balanced computation case: completion time (upper) and efficiency (lower)

show the actual advantages and limits of the approach. For the experiments we used several different codes including:

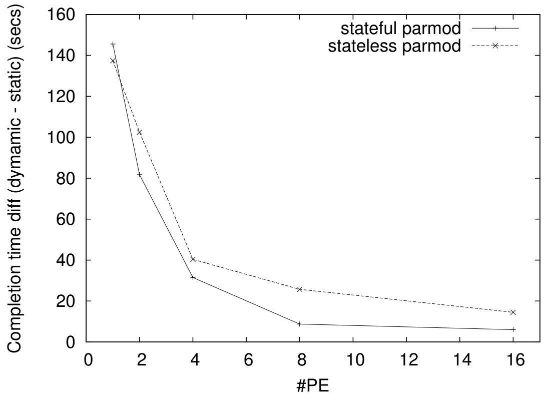
- **Mandelbrot** We considered two codes computing the Mandelbrot set. The first one is a pure task farm code. The point coordinates (a set of point coordinates, actually) are passed to virtual processors. The task farm is implemented using a `parmod` with topology `none` [19]. The second code is programmed as a data parallel `parmod` with fixed stencil. This second version has been used to emulate unbalanced computations. The static virtual processor to virtual processor managers (i.e. processing elements) mapping implemented in standard ASSIST generates sensibly unbalanced computations in this case.
- **Jacobi** This is kind of differential equation solver code. It is implemented using a data parallel `parmod` with and fixed stencil (each virtual processor accesses state from its immediate, fixed neightborhood).
- **Shortest Path** In this case the code uses a data parallel `parmod` with variable stencil (the neighborhood accessed varies with the iteration variable value) to implement Floyd-Warshall algorithm.
- **Game of Life** In this case the code uses a data parallel `parmod` such as the one used in Jacobi, that is fixed stencil. However, due to the highly dynamic nature of the problem, the load of the virtual processors changes a lot during the computation.

The computational grains taken into account in the sample programs used in the experiments are $O(10)$, roughly. In other words, the time spent to compute a single virtual processor is about 10 times the time spent to transfer input and output data items to and from the processing element actually computing the virtual processor. Larger grain values obviously make things go better, in terms of efficiency. Smaller grain values increase the overhead so much that parallel execution is no more convenient with respect to sequential execution.

All the experiments have been performed on a 24 Pentium III 800MHz RLX blade chassis with Fast Ethernet interconnection. Actually, some processing elements have been used for the support processes (e.g. the interpreter and collector process of Figure 3 plus the task pool process), one was dedicated for blade cluster management. Therefore we had a maximum of 20 processing elements available and we decided to run experiments with power of 2 processing elements. Nothing in both the standard ASSIST implementation and in the MPI implementation prevents from using a number of processing elements which is not a power of 2.

Figure 4 shows the ”good case” with respect to the dynamic `parmod` implementation. In this case we used a computation where a significant fraction of the virtual processors computed for a very small amount of time before actually producing the result, while the other virtual processors computed for a very long time. The code used is the data parallel `parmod` implementation of Mandelbrot. A matrix of 160000×125 points has been used, with each virtual processor “holding” 1000 points, but similar results have been achieved also using 10000×1000 matrixes resulting in a larger number of smaller point partitions, i.e. with a smaller computation grain. In this case, the advantage of the dynamic implementation is large. As ready virtual processors are assigned to idle workers, load balancing is automatically achieved, which is not the case when static virtual processor assignment is used (current ASSIST implementation). In the Figure, the x-axis plots the number of processing elements used to compute the program while the y-axis plots the completion time in milliseconds.

Figure 5 is about the ”bad case” with respect to the dynamic `parmod` implementation, that is, it is relative to a data parallel computation with balanced load. The virtual processors in the `parmod` share a common state, in this application, which is an instance of the Jacobi code. In this case we used a 512×512 matrix, with 1 row per virtual processor. As the computational grain in this case was very low (a few floating point ops per cycle, we introduced a small delay loop computing some 2000 trigonometric operations in each cycle). The addition was performed in the sequential code, so it was equally increasing the computational grain of both the ASSIST and the MPI version of the application. Static partitioning of activities among the available processors works fine in this case, and therefore we expected to pay some sensible overhead due to the dynamic `parmod` implementation. Actually, the differences in computing times are not too much relevant. This demonstrates that the dynamic implementation of `parmod` is quite efficient.



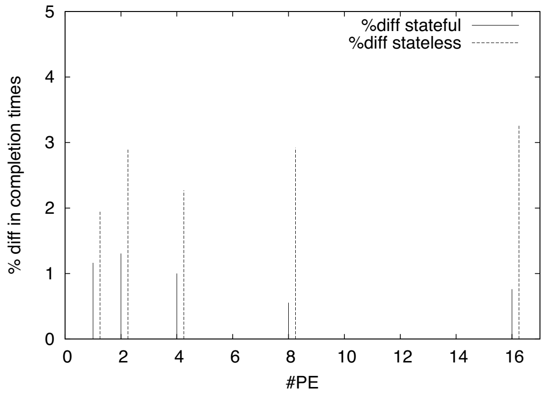
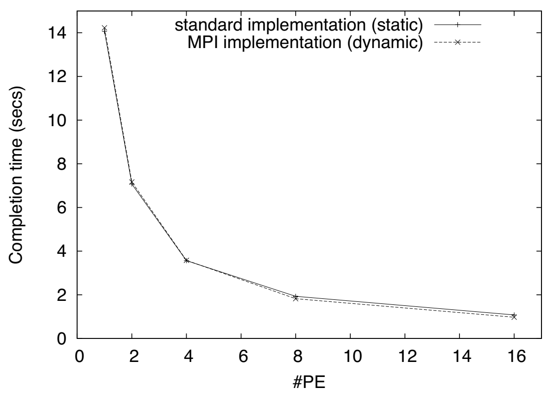


Figure 6. Dynamic (MPI `parmod`) vs. static (ASSIST) implementation of `parmod`: difference in completion times (absolute upper, percentage lower)

Figure 6 is again about a ”bad case” with respect to the dynamic `parmod` implementation, which is actually not so bad, as shown. In this case, we run two applications: both applications are plain data parallel application where each virtual processor computes for a comparable amount of time. One application does not share any kind of state among the virtual processors (the one marked as *stateless*) whereas the other one shares a state that has to be updated at each iteration (the one marked as *stateful*). Actually, the



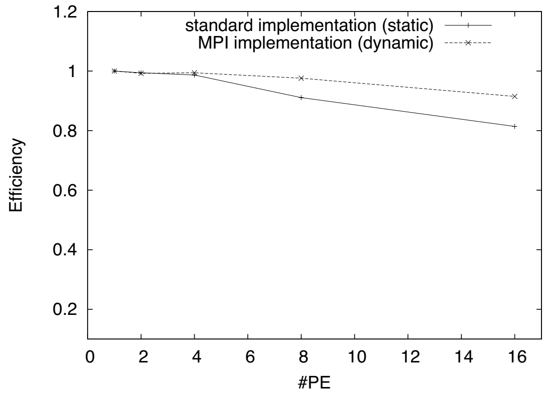


Figure 7. Dynamic (MPI `parmod`) vs. static (ASSIST) implementation of `parmod`: dynamic stencil (speedup upper, efficiency lower)

first one is an instance of the data parallel Mandelbrot and the second one is an instance of the data parallel Jacobi application. The two application instances have been calibrated in such a way they have a similar completion time, just playing on the dimensions of the matrixes involved. We expected to pay a penalty, in these cases, when using the MPI dynamic `parmod` library, as the static assignment works and it is paid just once and for all, in terms of overhead, while the fetching of ready virtual processors is paid many times in the dynamic implementation. Figure 6 upper plot shows the absolute differences in the completion times of the two data parallel applications with standard, static and MPI, dynamic implementations. Figure 6 lower plot shows the same difference in percentage. The overhead of the dynamic implementation (i.e. the increase in completion time) is always below 5%, which is definitely not bad.

Figure 7 is about the Shortest Path application. In this case, the stencil used to access the remote shared state variables in the `parmod` varies at each iteration of the virtual processor computations. The differences between the completion times of the standard, static implemention and the MPI, dynamic one are relatively small, but the MPI implementation is more efficient than the standard ASSIST one.

5 Conclusions

We outlined a new implementation schema for the ASSIST `parmod` generic parallel skeleton. The new implementation is completely dynamic in that logical parallel activities are mapped to available processing elements in a completely dynamic way, rather that statically, adopting some kind of block/cyclic distribution. The new implementation schema has been implemented in a C++/MPI library that allowed making different experiments comparing ”dynamic” code with actual, ”static”, ASSIST code. The results show that in case static assignment of virtual processors to processing elements does not guarantee load balancing, the performance of our dynamic implementation is *far* better. And this was the expected result, of course. What was not expected is that in those cases where the static assignment guarantees load balancing, the dynamic implementation does not add a significant overhead and execution times are comparable with those achieved with the classical ASSIST implementation of `parmod`. This opens new perspectives in the development of ASSIST. In particular, the macro data flow implementation of skeleton based programming environments has already been experimented both in our group [11, 3] and in the Skipper project [17, 16], but this is the very first time we use it to implement a complex, stateful skeleton, rather than simpler, stateless skeletons.

6 Acknowledgements

This work has been partially supported by Italian national FIRB project no. RBNE01KNFP *GRID.it*, by Italian national strategic projects *legge 449/97* No. 02.00470.ST97 and 02.00640.ST97, and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] The MANIFOLD home page, 2002. http://www.cwi.nl/projects/manifold.

[2] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzoloand M. Torquati, M. Vanneschi, and C. Zoccolo.

The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of Intl. Conference EuroPar2003: Parallel and Distributed Computing*, number 2790 in LNCS. Springer, 2003.

[3] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[4] Marco Aldinucci, Marco Vanneschi, Corrado Zoccolo, Massimo Torquati, Luca Veraldi, and Alessandro Petrocelli. Dynamic Reconfiguration of Grid-aware applications in ASSIST. In J. Cunha, editor, *Proceedings of Euro-Par 2005*. Springer Verlag, September 2005. Lisboa.

[5] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[6] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, Dec. 1999.

[7] R. Baraglia, M. Danelutto, D. Laforenza, S. Orlando, P. Palmerini, R. Perego, P. Pesciullesi, and M. Vanneschi. AssistConf: A Grid Configuration Tool for the ASSIST Parallel Programming Environment. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 193–200. IEEE, 2003.

[8] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[9] M. Cole and A. Benoit. The Edinburgh Skeleton Library home page, 2005. http://homepages.inf.ed.ac.uk/abenoit1/eSkel/.

[10] M. Danelutto. Dynamic Run Time Support for Skeletons. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, editors, *Proceedings of the International Conference ParCo99*, volume Parallel Computing Fundamentals & Applications, pages 460–467. Imperial College Press, 1999.

[11] M. Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[12] M. Danelutto and L. Rovetti. Distributed task pool implementation. In *Proceedings of the VECPAR'2002, 5th International Meeting on High Performance Computing for Computational Science, Part II*, pages 495–506, June 2002. Oporto, Portugal.

[13] H. Kuchen. A skeleton library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. "Springer" Verlag, August 2002.

[14] S. MacDonald, J. Anvik, S. Bromling, J. Scaheffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Computing*, 28(12), 2002.

[15] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison Wesley, 2004.

[16] J. Serot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 2002.

[17] J. Serot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel computing*, 28(12):1685–1708, 2002.

[18] D. Szafron and J. Schaeffer. An experiment to measure the usability of parallel programming systems. *Concurrency - Practice and Experience*, 8(2):147–166, 1996.

[19] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications . *Parallel Computing*, 12, December 2002.

IEEE COMPUTER SOCIETY