# Towards Fully Adaptive Pipeline Parallelism for Heterogeneous Distributed Environments

**Citation for published version:**
González-Vélez, H & Cole, M 2006, Towards Fully Adaptive Pipeline Parallelism for Heterogeneous Distributed Environments. in *ISPA.* LNCS, Springer-Verlag GmbH, pp. 916-926.
https://doi.org/10.1007/11946441_82

**Digital Object Identifier (DOI):**
10.1007/11946441_82

**Link:**
Link to publication record in Edinburgh Research Explorer

**Document Version:**
Publisher's PDF, also known as Version of record

**Published In:**
ISPA

# Towards Fully Adaptive Pipeline Parallelism for Heterogeneous Distributed Environments

Horacio González-Vélez and Murray Cole

Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, UK
h.gv@ed.ac.uk, mic@inf.ed.ac.uk

**Abstract.** This work describes an adaptive parallel pipeline skeleton which maps pipeline stages to the best processors available in the system and clears dynamically emerging performance bottlenecks at run-time by re-mapping affected stages to other processors. It is implemented in C and MPI and evaluated on a non-dedicated heterogeneous Linux cluster. We report upon the skeleton's ability to respond to an artificially generated variation in the background load across the cluster.

## 1 Introduction

Pipelining is the decomposition of a repetitive sequential process into a succession of distinguishable sub-processes called *stages*, each of which can be efficiently executed on a distinct processing element or elements which operate concurrently.

In software, this approach is widely used to address grand-challenge computational science problems [1], numerical linear algebra algorithms [2], and signal processing applications [3]. Pipelines are exploited at fine-grained level in loops through compiler directives and in operating system file streams, and at coarse-grained level in parallel applications employing multiple processors. In particular, coarse-grained pipeline applications refine complex algorithms into a sequence of independent computational stages where the data is "piped" from one computational stage to another. Each stage is then allocated to a processing element in order to compose a parallel pipeline. Our pipeline follows this model.

The performance of a pipeline can be characterised in terms of *latency*, the time taken for one input to be processed by all stages, and *throughput*, the rate at which inputs can be processed when the pipeline reaches a steady state. Throughput is simply related to the processing time of the slowest stage, or *bottleneck*. When handling a large number of inputs, it is throughput rather than latency which constrains overall efficiency. Our system schedules and dynamically reschedules stages to processors, in the face of the dynamically varying processor capability which is typical of grid systems, with a view to maintaining high throughput.

The problem addressed in this paper is as follows: given a parallel pipeline program, find an effective way to improve its performance on a heterogeneous distributed environment by adapting dynamically to external load variations.

M. Guo et al. (Eds.): ISPA 2006, LNCS 4330, pp. 916–926, 2006.

Our adaptive parallel pipeline has two main components: calibration and feedback. Initially the calibration is used to map stages to the best processors available in the system. Subsequently, the feedback mechanism clears performance bottlenecks at run-time by re-mapping the stages to other processors. This pipeline is implemented as a stateless skeleton in C and MPI. We present promising results of parallel executions in a non-dedicated heterogeneous Beowulf cluster, using a stage function based on a numerical benchmark.

This paper is structured as follows. First, we provide motivation for this work. Then we describe the adaptive parallel pipeline algorithm and its implementation, followed by the experimental evaluation. Finally we discuss some related approaches and make final remarks.

## 2 Motivation

### 2.1 Idealised Pipelines

We must review some generic performance issues in pipelined processing. Suppose that the original sequential process requires time $t_s$ to process a single input. Consider an $n$-stage pipeline, in which $t_\imath$ is the execution time for the $\imath^{th}$ stage.

In an idealised model, without significant communication costs, the sequential and parallel (one processor per stage) times to process $S$ inputs are then

$$\begin{aligned} T_{seq} &= S \times t_s \\ T_{par} &= \sum_{\imath=1}^{n} t_\imath + (S-1) \times \max(t_\imath) \\ &\text{where } \max(t_\imath) \text{ is the bottleneck stage time} \end{aligned}$$

It is well known that perfect pipelined performance is obtained when the stage times $t_\imath$ are all equal to $\frac{t_s}{n}$, since we can then reduce the expressions to:

$$\begin{aligned} T_{seq} &= S \times t_s \\ T_{par} &= t_s + (S-1) \times \frac{t_s}{n} \end{aligned}$$

so that as $S$ grows large, speed-up asymptotically approaches $n$.

Outside this perfect situation, it is more important to reduce the bottleneck time than the latency, since the former affects the multiplicative term in $T_{par}$, where the former affects only the asymptotically insignificant additive term.

### 2.2 Pipelines on Dynamically Heterogenous Resources

With the advent of heterogeneous distributed systems, whether geographically-contiguous (clusters) or in different administrative and geographical domains (grids [4]), it is widely acknowledged that one of the major challenges in programming support is the prediction and improvement of performance [5]. Such systems are characterised by the dynamic nature of their heterogeneity, due to shifting patterns in background load which are not under the control of the individual application programmer. The challenge is therefore to produce and support applications which can respond to this variability.

In our current work we focus on the computational aspects of heterogeneity. In what follows we make assumptions designed to keep the number of experimental variables tractable. Subsequent work will consider relaxation of these assumptions. Specifically, we assume that

- the computational weight of each stage is identical, in the sense that all stages would take the same time to process one item if executed on the same reference processor. In effect, this is to assume that the programmer has done a good abstract job of balancing stages and reducing the bottleneck. This allows us to focus on addressing issues which arise when the available processors vary dynamically in performance with respect to such a reference processor.
- communication time is not significant. Note that this is not to assume that communication is negligible, but rather to assume that communication costs hinder all stages equally.

The challenge can now be stated simply. The application programmer is required to write sequential code for the body of each pipeline stage and make a call to our pipeline skeleton to apply these stages to a set of inputs. The "grid" provides a pool of available processors. Our system maps the stages to (a subset of) the processors. It may choose to map several stages to the same processor when this processor is more powerful than the others. Periodically, our system checks the progress of the computation and may decide to remap some or all of the stages. In the following section we describe the mechanisms employed to construct this overall framework.

## 3 Adaptive Pipeline Parallelism

The core of our system is an algorithm for mapping pipeline stages to processors. Its main feature is that processors are calibrated at run-time, to provide the performance information upon which the mapping is based. The mapper is embedded within an iterative scheduling scheme, allowing the pipeline implementation to be adapted to prevailing conditions within the pool of available processors. We will now discuss the mapper and the rescheduler in turn.

### 3.1 Mapping Stages to Processors

The first step in mapping is to determine the current 'fitness' of each available processor. This is achieved by running, by way of calibration, an instance of one of the stage functions on each processor, and measuring the execution time. Any stage will do, since we have assumed that all stages are equally inherently 'heavy'. This allows us to rank processors by descending fitness (i.e. by increasing calibration time). We can immediately discard all but the $n$ fittest processors from the initial mapping. The mapping is generated by a greedy algorithm, which computes $x_i$, the number of stages to be executed by processor $i$.

Algorithm 1 provides a detailed description of the calibration procedure. In this algorithm, $X$ is the aggregated array containing all $x_i$ entries, $t$ records the aggregated execution time of the stage function in every node, and *Chosen* is the array of selected nodes. That is to say, $P$ contains all nodes in the `MPI_COMM_WORLD`, *Chosen* holds only the processors to be used, and $X$ indicates the number of processes per node capped by the maximum processes per node. As per the greedy nature of the algorithm, every entry $t_i$ in $t$ takes into account the workload generated by the increasing number of stage-function instances to be executed in a given node.

We make an initial mapping in which one stage is assigned to each of the $n$ fittest processors. We construct a ranking of the chosen processors, according to the time $t_\imath$ they will take to process an item, given their currently allocated stages. Initially this is identical to the calibration ranking, but as a processor is assigned extra stages, its $t_\imath$ will be the product of the number of allocated stages and its original calibration time.

The initial mapping is iteratively improved by application of the following step:

> *Consider the effect of moving one stage from the processor $P_b$ with the highest processing time (i.e. the current bottleneck) to the processor $P_l$ with the lowest processing time. If the new resulting processing time at $P_l$ is smaller than the original processing time at $P_b$ then make the switch.*

Iteration proceeds until no further improvement is possible. It is not difficult to see that such a strategy is optimal.

Suppose there was a better mapping $M'$ than the one with which the algorithm terminates, $M$. Suppose that the bottleneck processor $P_b$ in $M$ is assigned $k$ stages. $M$ must assign fewer than $k$ stages to $P_b$ (otherwise it wouldn't better) and more stages to at least one other processor $P_x$ (because the extra stage must be assigned somewhere). Then $M$ can be improved by moving one stage from $P_b$ to $P_l$ (which may or may not be $P_x$, it doesn't matter), and the greedy algorithm will do so, thereby contradicting the (premature) assumption of termination. ∘

### 3.2 Monitoring Performance and Rescheduling

Once the pipeline is in operation, the feedback phase detects performance bottlenecks by checking whether all processors are functioning according to the initial calibration. Each stage times itself and propagates its current $t_i$ through the pipeline, piggy backed with the real data. The final stage verifies that the $T_{par}$ is acceptable by comparing with the original calibration times, using a fixed threshold to determine acceptability. This threshold regulates the margin before a re-calibration takes place and is expressed as a fraction of the original value.

**Algorithm 1.** Calibration Algorithm

```
Data: $f$: Stage Functions;
$n$: Number of Stages;
$P$: Nodes;
Result: $Chosen$: Lookup table of fittest processing elements;
$X$: Number of processes per $Chosen$ node;


forall nodes in $P$ do
    Execute $f$ concurrently ;
    Set $t_i \leftarrow$ execution_time($f$);
end
if root node then
    set $X \leftarrow 0$ ;
    collect $t_i$ into $t$;
    sort the $P$ nodes ; /* Using $t$ as key */
    set $\imath \leftarrow 0$ ;
    set $\ell \leftarrow n$ ;
    while $\imath < \ell - 1$ do
        set $flag \leftarrow false$ ;
        set $k \leftarrow \imath + 1$ ;
        while $k < \ell \ \wedge \ \neg flag$ do
            set $\alpha \leftarrow \lfloor \frac{t_k - t_i}{t_i / x_i} \rfloor$;
            if $\alpha \neq 0$ then
                set $t_i \leftarrow t_i + t_i / x_i$ ;
                set $x_i \leftarrow x_i + 1$ ;
                set $\ell \leftarrow \ell - 1$ ;
                insert_in_order $(t_i, t)$;
                set $flag \leftarrow true$ ;
            end
            if $\neg flag$ then
                $\imath \leftarrow \imath + 1$;
            end
        end
    end
    send $Chosen$ and $X$ to other nodes
else
    send time from this node to root node;
    receive $Chosen$ and $X$;
end
```

For example, if a threshold is equal to $X$, it indicates that if a $t_i$ is more than $(1 + X)$ times slower than the original worst recording, a bottleneck has been detected and a remapping is scheduled. Similarly, should the worst time be $(1 - X)$ times faster than the original best, remapping is considered.

It is crucial to note that the threshold is key to the adaptivity mechanism since a small value may cause too many remappings (thrashing) while a large one will deactivate the adaptiveness. Once a decision to remap has been made, the pipeline is allowed to drain before resuming under the new mapping.

## 4 Implementation

To facilitate our experiments we have designed an algorithmic skeleton [6], programmed in ANSI C with MPI to handle internode communication. Its API is

```
void pipeline(stage_t *stages, int no_stages,int in_data[], MPI_Comm comm)
```

The first parameter `stages` is an array of pointers to functions which contains the $f$ stages, `no_stages` is $n$, `in_data` is the input data stream $S$ (using simple integers here, but easily generalisable), and `comm` is a communicator encompassing the processor pool.

Based on previous experiences with skeletons on geographically dispersed grids [7], where we empirically learned the costly implications of the inherent synchronisation in collectives, we have based this design on explicit send-receive pairing. Internally, each stage is composed by a `MPI_Recv` call, the invocation to the $f$ function, and a `MPI_Send` call.

Internally, the processor pool is represented as a lookup table of active processors. Each processor uses the table to determine its predecessor and successor. It is built during the calibration process by simply sorting the processors in the communicator by execution times. The table is also of particular importance during process migration since the migration is in essence an exchange of its entries.

On the infrastructure side, there exist a few libraries which provide MPI process migration, mainly devoted to preserving index and context variables in loops. Although they address generic MPI programs, their use requires specialised underlying distributed filesystems [8] or daemon-based services [9]. Since a key criterion to the process migration is fast process migration, we opted to develop a simple process migration mechanism based on the aforementioned process pool.

It is important to stress the fact that there are not pre-determined processors required for the execution of the pipeline. That is to say, after calibration not even the `MPI_COMM_WORLD` root process must belong to the set of fittest processes. Therefore, a re-mapping always implies a process migration with result preservation.

## 5 Results

The full system has been compiled with gcc 3.4.4 using "-pedantic -ansi -Wall -O2" flags and employs LAM/MPI 7.1.1. It has been deployed on a non-dedicated heterogeneous Beowulf cluster, located in the School of Informatics of the University of Edinburgh, and configured as shown in Table 1. The processors have different frequencies and exhibit different performance as determined by the BogoMips reading which is the standard Linux benchmark.

For reproducibility purposes, we have employed as stage function the `whetstones` procedure from the 1997 version [10] of the Whetstone benchmark with parameters `(256,100,0)`. It accounts for some 5 seconds of double-precision floating-point processing on an empty node in the Beowulf cluster.

**Table 1.** Beowulf cluster `bw240` Configuration

| | |
|---|---|
| Nodes: | 64 |
| CPU: | Intel P4 |
| Memory: | 1 GB / node |
| Network: | 2x100Mb/s (Shared) |
| OS: | Linux Red Hat FC3 - Kernel 2.6 |
| BogoMips: | 3350.52–3555.32 |

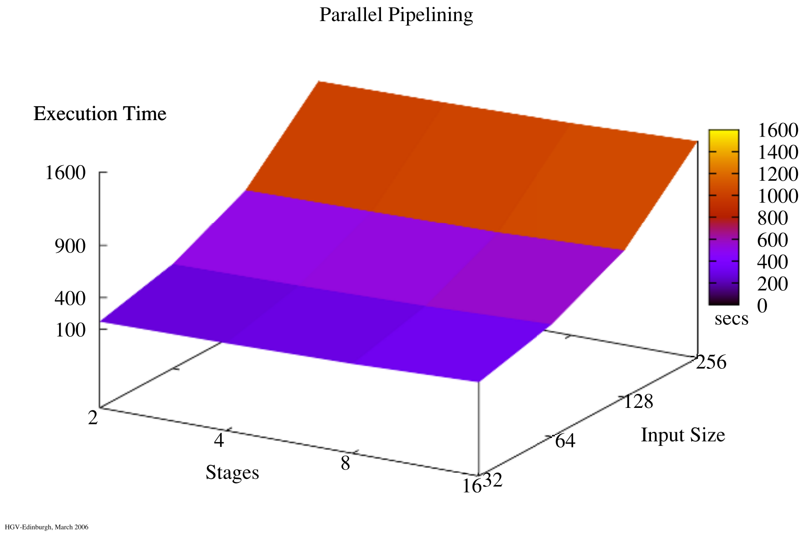


**Fig. 1.** Correlation between the size of the data input $S$ and the number of stages $n$

Thus, all variability in the system is due to external load and, to a lesser extent, to the difference in performance among processors. All execution times reported below sum up the average of three measurements, with a standard deviation of less than 1%.

Figure 1 shows a "sanity-check" initial exploration of the parameter space, running pipelines with 2, 4, 8 and 16 identical stages, one per processor (note that a pipeline with more stages is doing more work in absolute terms) on increasingly large inputs. The execution times are primarily determined by $|S|$, the input size of the data stream, and marginally influenced by the number of stages $n$ in the pipeline.

We have firstly explored the overhead incurred by the calibration phase. Figure 2 shows that the overhead is minimal and increases at a slow rate ($< 1\%$ for every power-of-two increment in the number of processes).

We then measured the performance impact of our system under different load conditions. Figure 3(a) depicts the times of different executions using two

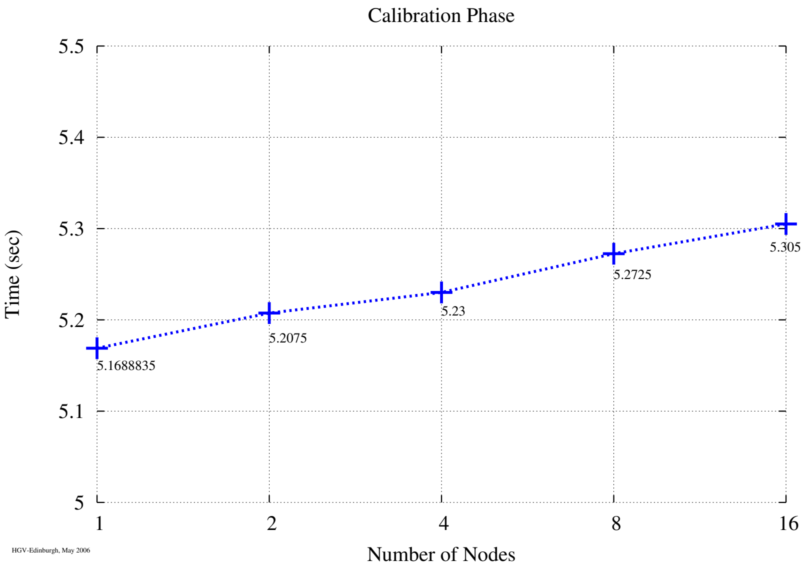
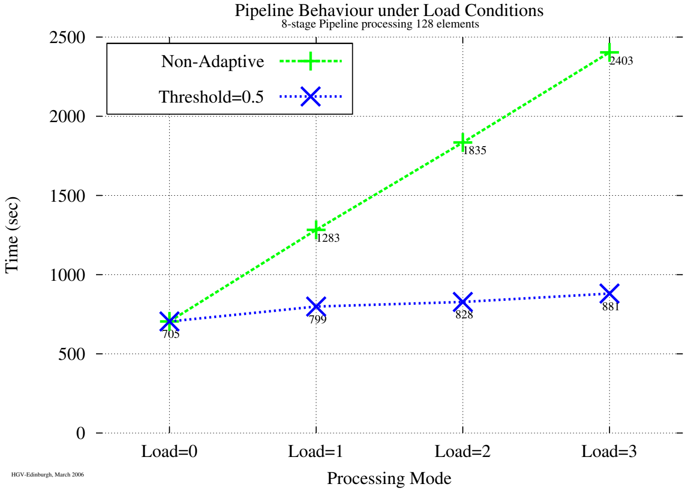
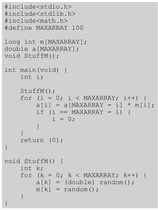


**Fig. 2.** Calibration phase execution times for 1,2,4,8, and 16 processes



(a)



```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#define MAXARRAY 100

long int m[MAXARRAY];
double a[MAXARRAY];
void StuffM();

int main(void) {
    int i;

    StuffM();
    for (i = 0; i < MAXARRAY; i++) {
        a[i] = a[MAXARRAY - i] * m[i];
        if (i == MAXARRAY - 1) {
            i = 0;
        }
    }
    return (0);
}

void StuffM() {
    int k;
    for (k = 0; k < MAXARRAY; k++) {
        a[k] = (double) random();
        m[k] = random();
    }
}
```

(b)

**Fig. 3.** (a) Comparison of the parallel pipeline using two threshold parameters: Infinite (non-adaptive) and 0.5. (b) Load generating program (adapted from [11]).

threshold parameters: infinite, which implies a non-adaptive pipeline, and 0.5. This threshold value was empirically determined using a series of test runs. Figure 3 (b) shows the actual program employed to generate load.

Taking into account the fair CPU allocation algorithm used in Linux and to assure the existence of changing load conditions, we have incrementally injected load dynamically to the system using a simple load generation program. Each instance of this program added 1 to the load displayed by the Linux `uptime` command in a certain node ("bottleneck node") until this node became a bottleneck, while the rest of the processors did not experience any significant load variation. Thus a $Load = 0$ implies no bottlenecks, $Load = 1$ an instance of the load generator was running on the "bottleneck node" and so on. The instances were triggered after 60 seconds from the start of the program.

Figure 3(a) shows a comparison of the measured execution times with $n = 8$ and $|S| = 128$. The x-axis indicates the injected load, i.e. the number of instances of the load generator running in one processor, which were triggered during the pipeline operation.

We see that the adaptive methodology has responded well under changing load conditions, since the execution times in the non-adaptive parametrisation have increased at a considerably higher rate than the adaptive ones.

## 6 Related Work

The scheduling problem of the parallel pipeline construct has been previously studied in the literature.

The LLP system [12] furnishes a conceptual framework for static multi-stage allocation using algorithmic skeletons. By approaching the problem with a 0/1 knapsack problem methodology, LLP is employed to develop a theoretical solution to stage scheduling.

Based on direct-acyclic graphs, the macro-pipelining methodology [13] gives a theoretical framework for scheduling parallel pipelines. While macro-pipelining provides guidance on the coarse distribution of work to different stages, its approach is limited to dedicated digital-signal processing systems.

Another approach presents a multi-layer framework for the stage scheduling in dedicated real-time systems [14]. This work describes a series of steps to calculate end-to-end latencies based on a time-series model for a video-conferencing application. Unfortunately, it does not address the general case.

Recent work on adaptive systems [15,16] has reinforced the importance of platform adaptation for optimisation of parallel codes in heterogeneous distributed systems. While implementation of parallel pipelines can be found in several established skeletal libraries [17,18], adaptive skeletal constructs have recently started to use resource-aware mechanisms based on process algebra [19] and statistical methodologies [20], paving the way to the development of a comprehensive library of self-adaptive algorithmic skeletons for non-dedicated heterogeneous systems.

## 7 Conclusions

Our methodology is fundamentally different since it provides a generic system

- to pragmatically tune up the pipeline parallelism skeleton regardless of the complexity of the stage functions (calibration phase); and

- to dynamically adapt to non-dedicated heterogenous environments once the pipeline processing is established (feedback).

A close examination of the methodology will show that there is certainly room for a more instrumented approach to the determination of the re-calibration threshold. Our work provides evidence that the proposed adaptive methodology enhances pipeline parallelism performance: execution times are almost an order of magnitude greater when not using the adaptive pipeline.

It is important to emphasise this work has covered load variations attributable to different processing capabilities, while maintaining the stage function complexity constant. Although this scenario does not comprehensively address all possible pipeline applications, it certainly provides guidance on the behaviour of the general case on distributed systems.

In time, we intend to expand the experiment space by analysing pipelines with stage functions with distinct complexity and, possibly, including a pipeline-oriented application such as image processing.

In the same manner, we will study the correlation between the threshold and the stage functions. Such a study may eventually lead to the automatic determination of the optimal threshold for a given set of stages.

## Acknowledgements

This work has been partly supported under the Chevening–Conacyt grant no. 81887. Horacio González–Vélez would like to thank Mary Cryan of the Laboratory for Foundations of Computer Science in Edinburgh for the early discussions on the calibration algorithm.

## References

1. Pancake, C.M.: Is parallelism for you? IEEE Computat. Sci. Eng. **3** (1996) 18–37
2. Heller, D.: A survey of parallel algorithms in numerical linear algebra. SIAM Review **20** (1978) 740–777
3. Fleury, M., Downton, A.: Pipelined Processor Farms: Structured Design for Embedded Parallel Systems. Wiley-Interscience, New York (2001)
4. Foster, I., Kesselman, C., eds.: The Grid: Blueprint for a new computing infrastructure. Second edn. Morgan Kaufmann, San Francisco (2003)
5. Laforenza, D.: Grid programming: some indications where we are headed. Parallel Comput. **28** (2002) 1733–1752
6. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, London (1989)
7. González-Vélez, H.: An adaptive skeletal task farm for grids. In: Euro-Par 2005. Number 3648 in Lect. Notes Comput. Sc., Springer-Verlag (2005) 401–410
8. Vadhiyar, S.S., Dongarra, J.: SRS: A framework for developing malleable and migratable parallel applications for distributed systems. Parallel Process. Lett. **13** (2003) 291–312
9. Sievert, O., Casanova, H.: A simple MPI process swapping architecture for iterative applications. Int. J. High Perform. Comput. Appl. **18** (2004) 341–352

10. Longbottom, R.: C/C++ Whetstone benchmark single or double precision. ftp.nosc.mil/pub/aburto (1997) Official version approved by H. J. Curnow on 21/Nov/97.
11. Gunther, N.J.: Analyzing Computer System Performance with Perl: PDQ. Springer-Verlag, Berlin (2004)
12. González, D., Almeida, F., Moreno, L.M., Rodríguez, C.: Towards the automatic optimal mapping of pipeline algorithms. Parallel Comput. **29** (2003) 241–254
13. Banerjee, S., Hamada, T., Chau, P.M., Fellman, R.D.: Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. IEEE Trans. Acoust. Speech Signal Process. **43** (1995) 1468–1484
14. Chatterjee, S., Strosnider, J.K.: Distributed Pipeline Scheduling: A framework for distributed, heterogeneous real-time system design. Comput. J. **38** (1995) 271–285
15. Vadhiyar, S.S., Dongarra, J.J.: Self adaptivity in grid computing. Concurrency Computat. Pract. Exper. **17** (2005) 235–257
16. Kelly, P.H.J., Beckmann, O.: Generative and adaptive methods in performance programming. Parallel Process. Lett. **15** (2005) 239–255
17. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Gener. Comput. Syst. **19** (2003) 611–626
18. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. **30** (2004) 389–406
19. Yaikhom, G., Cole, M., Gilmore, S.: Combining measurement and stochastic modelling to enhance scheduling decisions for a parallel mean value analysis algorithm. In: ICCS 2006. Number 3992 in Lect. Notes Comput. Sc., Springer-Verlag (2006) 929–936
20. González-Vélez, H.: Self-adaptive skeletal task farm for computational grids. Parallel Comput. (2006) In press doi:10.1016/j.parco.2006.07.002.