An Adaptive Parallel Pipeline Pattern for Grids

Horacio González–Vélez
Institute for Computing Systems
Architecture, School of Informatics,
University of Edinburgh, UK
e-mail: h.gv@ed.ac.uk

Murray Cole
Institute for Computing Systems
Architecture, School of Informatics,
University of Edinburgh, UK
e-mail: mic@inf.ed.ac.uk

Abstract

This paper introduces an adaptive parallel pipeline pattern which follows the GRASP (Grid-Adaptive Structured Parallelism) methodology. GRASP is a generic methodology to incorporate structural information at compile time into a parallel program that enables it to adapt automatically to dynamic variations in resource performance. GRASP instruments the pipeline with a series of pragmatic rules, which depend on particular performance thresholds based on the computation/communication patterns of the program and the availability of resources in the grid. Our parallel pipeline pattern is implemented as a parameterisable C/MPI API using a variable-size input data vector and a stage function array. We have evaluated its efficiency using a numerical benchmark stage function in a non-dedicated computational grid environment.

Keywords: Algorithmic Skeletons; Parallel Patterns; Parallel Programming; Structured Parallelism; Grid Computing; Grid Resource Management

1. Introduction

Ranked as the third most influential parallel and distributed concept of the past millennium—behind Amdahl’s law and the Internet [27], pipelines are well-known in the literature. They enable the decomposition of a repetitive sequential process into a succession of distinguishable subprocesses called stages, each of which can be efficiently executed on a distinct processing element or elements which operate concurrently.

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. With application in grand-challenge computational problem solving, numerical linear algebra, signal/image processing and grid workflows, coarse-grained pipelines refine complex algorithms into a sequence of independent computational stages where the data is “piped” from one computational stage to another. Each stage, composed by a simple consumer, a computational function, and a simple producer, 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 primarily influenced by 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, since the latency is only relevant to measure the time to fill up the pipeline initially. Once full, the pipeline steadily delivers results at the throughput ratio. Hence, in order to improve the overall efficiency of a parallel pipeline, one requires to minimise the bottleneck processing time.

Meanwhile, the use of efficient programming models and structures, which can be staged in a scalable structured fashion, has long been sought after in computational grids [10]. These programming models must be necessarily performance-oriented, and they are expected to provide guidance on the overall execution of their jobs in order to assist in the deployment of heterogeneous resources and policies.

Hence, the problem addressed in this paper is as follows: given a parallel pipeline program, find an effective way to improve its performance in a computational grid environment by effectively mapping the pipeline stages to the best available processors and adapting dynamically to external load variations.

We propose an adaptive parallel pipeline pattern which follows the GRASP (Grid-Adaptive Structured Parallelism) methodology [17]. GRASP is a generic methodology to incorporate structural information at compile time into a parallel program that helps it to adapt at execution time. GRASP instruments the program with a series of pragmatic rules,
which depend on particular performance thresholds based on the computation/communication patterns involved and the availability of resources in the grid. Each rule helps to determine and amend the schedule. This paper significantly extends our initial approach [16] by deploying GRASP in order to efficiently schedule a pipeline.

Our parallel pipeline pattern is implemented as a parameterisable function call using a variable-size input data vector and a stage function array. We have evaluated its efficiency using a numerical benchmark stage function in a non-dedicated computational grid environment.

This paper is structured as follows. Firstly, section 2 provides the motivation for this case study. Secondly, section 3 describes an adaptive pipeline parallelism approach as an instantiation of the GRASP methodology. Thirdly, section 4 describes the API implementation, followed by the experimental evaluation in section 5. Finally, section 6 lists related approaches to this work, and section 7 presents our conclusions.

2. Motivation

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication and interaction [6, 7]. Skeletons provide top-down design, composition, and control inheritance throughout the whole program structure. Parallel programs are expressed by interweaving parameterised skeletons analogously to the way sequential structured programs are constructed. Skeletons provide a clear and consistent behaviour across platforms, with the underlying structure depending on the particular implementation [24].

By decoupling the detail from the structure of a parallel program, they benefit entirely from any performance improvements in the system infrastructure while preserving the final result. Such decoupling has allowed skeletons to be efficiently deployed on different dedicated and non-dedicated architectures including symmetric multiprocessing, massively parallel processing, clusters, constellations, and grids [8]. The “Berkeley View” whitepaper has only reinforced the importance of not only producing realistic benchmarks for parallel programming models based on patterns of computation and communication, but also developing programming paradigms which efficiently deploy scalable task parallelism [2].

It is widely acknowledged that one of the major challenges in programming support for grids is the prediction and improvement of performance [20]. Grid 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. From a systems administration perspective, the demand for strategies which minimise communication overheads make resource management and scheduling key to the correct functioning of a computational grid.

In principle, it is expected that efficient parallel applications must be aware of grid conditions, and adapt execution according to variations in the available computation and communication resources. The challenge is therefore to produce and support applications which can respond automatically to this variability.

Based on the central premise of application adaptiveness to resource availability [19, 30], we would like to research their actual correlation by employing the forecasting information of a pipeline skeleton and fostering adaptiveness through its structure exploitation.

3. Methodology

GRASP is a generic methodology to incorporate structural information at compile time into a parallel program that helps it to adapt at execution time. It instruments the program with a series of pragmatic rules embedded in the algorithmic skeletons, which depend on particular performance thresholds based on the nature of the skeleton, the computation/communication ratio of the program, and the availability of grid resources. GRASP comprises a set of rules and their rationale to apply such methods to a set of skeletons, where every rule essentially constitutes a defined scheduling method.

GRASP can forecast and enhance the performance of a skeletal application in a grid by exploiting the knowledge of the skeletal structure while preserving the skeletal behaviour. The main difference between GRASP and other performance approaches is that it is intended to be oriented toward structured parallelism, adaptable by construct, and focused on empirical, system-infrastructure methodologies. GRASP has been successfully employed in a single-round scheduling task farm [15], yielding to interesting applications in computational cell biology [18].

GRASP comprises four phases: programming, compilation, calibration and execution. In the programming phase for the pipeline, the application programmer is required to write sequential code for the body of each pipeline stage. In the compilation phase, the programmer makes a call to the pipeline skeleton to apply these stages to a set of inputs. Then, during the calibration phase, the system maps the stages to (a subset of) the nodes in the grid and calculates a performance threshold. During the execution phase, our system periodically checks the progress of the computation and decides whether to remap some or all of the stages based on the performance threshold.

We will concentrate in this work on the instantiation of the GRASP calibration and execution phases in the context of our adaptive pipeline.

We have assumed that:
1. The computational complexity of each function executed in a stage is identical, in the sense that all stages would take the same time to process one item if executed on a dedicated reference processor. In effect, this presumes that the programmer has done a good abstract job of balancing stages and allows us to examine scheduling issues caused by dynamic variation in the underlying machine performance independently of static ones caused by the application itself. We designate this reference stage function as \( f \).

2. The 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, i.e., all consumer-producer pairing operate similarly.

### 3.1. Calibration

The purpose of this phase is twofold: calculate the stage-to-node mapping and determine the performance threshold which governs the feedback.

The assignment of stages to processors is widely known as stage mapping, or mapping, and finding the optimal mapping for a pipeline of any given length on a certain number of processors is the mapping problem [4, 5]. This problem is also referred to in the literature as pipeline scheduling as, in general, the scheduling of parallel tasks deals with the order in which tasks are executed and their assignment to nodes.

The mapping problem can be formally enunciated as follows. Let \( n \) be the number of stages in a pipeline and \( P \) the total number of processors of a distributed system. A mapping \( M \) is a pair \((\text{Chosen}, \vec{x})\) which complies with properties (1) and (2).

\[
\text{Chosen} \text{ is a finite set of processors } \text{Chosen} \subseteq P \quad (1)
\]

\[
\vec{x} \text{ is a vector of size } |\text{Chosen}| \text{ such that } \sum_{i=1}^{\text{Chosen}} x_i = n \quad (2)
\]

For our case, the first step in determining \( M \) consists in finding the fitness of each available processor. This is achieved by concurrently running an instance of \( f \) on each processor and measuring its execution time.

In practical terms, any stage will do, since we have assumed that all stages are equally representative in computational terms, making the execution time of this overall phase determined by the execution time of the slowest node. Note that if this were not the case, the overall complexity of the calibration would be greatly increased, as a result of the number of combinations of the stage functions and the fitness of every node, with the augmented complexity of the changing conditions of the underlying infrastructure. Therefore, by assuming that all the stage functions have similar complexity, the calibration phase produces an accurate snapshot of the fitness of all nodes at a given point in time.

The calibration 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 this initial mapping \( M \).

Formally, assume that \( M \) and \( t_i \) are the initial mapping and its calibration time for node \( i \) respectively. Properties (1) and (2) hold for \( M \) and, by construction:

\[
|\text{Chosen}| = n
\]

and \( p_6 \) the last node in \( \text{Chosen} \), is the bottleneck since:

\[
t_{i-1} \leq t_i \leq t_{i+1} \forall i \in \text{Chosen}
\]

Then, a greedy algorithm uses this initial (na"ive) mapping \( M \), in which one stage is assigned to each of the \( n \) fittest nodes, and iteratively tries to improve its stage allocation. It compares the impact of moving one stage from the processor \( p_6 \) to the processor \( p_1 \), the first one in the active mapping one and therefore the one with lowest processing time. If the new processing time at \( p_1 \) —the product of the number of its allocated stages plus one and its original calibration time—is smaller than the original processing time at \( p_6 \) then it makes the switch. It then re-ranks \( p_1 \) according to its new processing time. Iteration proceeds until no further improvement is possible. We call the resulting mapping \( M^\ast \). Let us illustrate this with a simple example.

**Example 1**: Assume that \( P = 32 \), \( n = 16 \), and \( \vec{t} \) stores the following 32 values entries corresponding to the calibration times of an equal number of nodes:

\[
p_1, \ldots, p_{32}:
\]

\[
\vec{t} = \{22, 13, 29, 16, 5, 11, 1, 19, 17, 29, 14, 31, 36, 15, 10, 25, 19, 18, 30, 18, 20, 24, 31, 25, 22, 23, 32, 29, 38, 19, 17\}
\]

Initially,

\[
M = \{\text{Chosen} = \{p_7, p_5, p_15, p_6, p_2, p_11, p_14, p_4, p_9, p_{32}, p_{18}, p_{20}, p_{17}, p_{31}, p_{21}\}, \vec{x} = \{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}\}
\]

After the application of the greedy algorithm,

\[
M^\ast = \{\text{Chosen} = \{p_5, p_{1}, p_6, p_7\}, \vec{x} = \{2, 1, 1, 1\}\}
\]

Table 1 shows the steps performed by the execution of the greedy algorithm.

In addition to the mapping, the calibration phase performs the calculation of the performance threshold, key to the adaptivity mechanism of the pipeline.

In the idealistic case of a totally-dedicated fully-homogeneous system, the threshold ought to be large, as the
<table>
<thead>
<tr>
<th>Chosen $\vec{t}$</th>
<th>$p_7$</th>
<th>$p_5$</th>
<th>$p_{15}$</th>
<th>$p_6$</th>
<th>$p_2$</th>
<th>$p_{11}$</th>
<th>$p_{14}$</th>
<th>$p_4$</th>
<th>$p_9$</th>
<th>$p_{32}$</th>
<th>$p_{18}$</th>
<th>$p_{20}$</th>
<th>$p_8$</th>
<th>$p_{17}$</th>
<th>$p_{31}$</th>
<th>$p_{21}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_7$</td>
<td>$p_5$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_7$</td>
<td>$p_5$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_5$</td>
<td>$p_7$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_5$</td>
<td>$p_7$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_5$</td>
<td>$p_7$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_5$</td>
<td>$p_7$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_5$</td>
<td>$p_7$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_5$</td>
<td>$p_7$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Chosen $\vec{t}$</td>
<td>$p_5$</td>
<td>$p_7$</td>
<td>$p_{15}$</td>
<td>$p_6$</td>
<td>$p_2$</td>
<td>$p_{11}$</td>
<td>$p_{14}$</td>
<td>$p_4$</td>
<td>$p_9$</td>
<td>$p_{32}$</td>
<td>$p_{18}$</td>
<td>$p_{20}$</td>
<td>$p_8$</td>
<td>$p_{17}$</td>
<td>$p_{31}$</td>
<td>$p_{21}$</td>
</tr>
<tr>
<td>$\vec{x}$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1. Execution Steps of the Greedy Algorithm
overall system will not benefit from a re-calibration since all nodes are equally fit. In a very dynamic system with highly heterogeneous nodes, the threshold must be small to allow rapid reaction to load changes while maintaining a balance to avoid unnecessary re-calibrations. A small threshold can cause too many re-mappings (thrashing) while a very large one effectively deactivates the adaptiveness. Arguably, it needs to be determined dynamically based on the fitness of all nodes in the system.

We propose a performance threshold for our adaptive pipeline as the inverse standard deviation of the calibration times of the nodes in the initial mapping. If we consider the differences between the times as a measure of the dispersion of the overall system, highly-dispersed—typically heterogeneous—systems have calibration times with a small threshold, which will trigger a node calibration with subtle variation in node performance. Conversely, more steady—typically homogeneous—systems present a large threshold which will not react as fast to variations in node performance.

Assuming a normal distribution of the calibration times, their standard deviation, \( \sigma \), can be calculated as:

\[
\sigma = \sqrt{\frac{1}{|\text{Chosen}|} \sum_{i=1}^{|\text{Chosen}|} (t_i - \bar{t})^2}
\]

where \(|\text{Chosen}|\) and \(\bar{t}\) are the number of nodes in the initial mapping \(M\) and the average calibration time respectively. As \(\sigma\) quantifies the dispersion of the calibration times, its inverse can arguably be used as a measure of adaptivity, e.g., using the execution times shown for the 16 nodes in \(\text{Chosen}\) from the initial \(M\) in example 1, \(\sigma = 5.404\) and \(\text{threshold} = 0.185\).

While the aforementioned procedure to automatically determine the threshold does not explicitly correlate the times with the actual node load, it provides an accurate steering criteria for adaptivity, as it is arguable that unchanging, either heavily- or lightly-loaded, systems will not benefit from a re-calibration.

In summary, the calibration phase performs the following steps:

1. records the execution time of the stage function in every node using \(\vec{t}\);
2. initialises \(M\) with the first \(n\) fittest nodes, sorted using \(\vec{t}\) as key and allocated one stage each;
3. calculates the \textit{threshold} as the inverse standard deviation of \(\vec{t}\);
4. employs a greedy strategy to determine the final mapping \(M_f\); and, finally,
5. broadcasts \(M_f\) and the \textit{threshold} to all nodes.

### 3.2 Execution

The purpose of this phase is to detect performance fluctuations in the pipeline and subsequently trigger re-mappings. Once the pipeline is in operation, this phase detects performance fluctuations 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 the performance threshold to determine acceptability. The threshold regulates the margin before a re-calibration takes place and is expressed as a fraction of the original value.

The principle is simple: assume that the threshold has been set at \(X\) during calibration and let \(t_i\) and \(t_b\) be the execution times for a certain stage \(i\) during normal operation and for the bottleneck at calibration respectively. By construction, \(t_b\) corresponds to the last node in the pipeline and is the worst time in the mapping. Thus, if

\[
(1 + X) \times t_i > t_b
\]

a performance fluctuation beyond the threshold has been detected and a re-calibration is scheduled. Once this decision has been taken:

1. the input stream is stopped and check-pointed,
2. the pipeline is allowed to drain,
3. the re-calibration takes place, and
4. execution is resumed under the new mapping.

Figure 1 illustrates schematically this procedure. While a re-calibration implies an overhead, we hypothesise that the performance gain may be superior as the pipeline throughput is determined by the bottleneck.

### 4. Implementation

Programmed in C with MPI, the pipeline pattern is abstracted as a parameterisable API:

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

This API works with the following parameters:

- \textit{stages} is an array of pointers to functions which contains the \(f\) stages;
- \textit{no\_stages} stores \(n\), the number of stages;
• \texttt{in\_data} is a vector representing the input data stream $S$; and

• \texttt{comm} is an MPI communicator encompassing $P$, the complete processor pool.

It is important to emphasise the API provides only the pipeline structure and its four externally-instantiated parameters regulate the pipeline behaviour, maintaining the philosophy of the skeletal paradigm.

We have based this design on explicit send-receive pairing. Internally, each stage is composed of an \texttt{MPI\_Recv} call, the invocation to the $f$ function, and an \texttt{MPI\_Send} call. The outputs from the pipeline are required to be explicitly written to a file by the last stage.

Pre-determined processors are not required for the execution of the pipeline. That is to say, after calibration, not even the \texttt{comm} root process must belong to the set of fittest processes, as a re-mapping always implies a full calibration.

The processor pool is represented as a lookup table of active processors. Referred to as \texttt{Chosen} previously, this table is built during the calibration process. It is also of particular importance during process migration, since the migration is, in essence, an exchange of its entries.

Since a key criterion is the provision of simple process migration with low overhead, we have opted for this simplistic swapping mechanism, in spite of the existence of generic libraries which provide MPI process migration. Mainly devoted to preserving index and context variables in loops, they address generic MPI programs, but their use requires specialised underlying distributed filesystems [29] or daemon-based services [25].

5. Evaluation

5.1. Grid Environment

The reported results have been obtained employing a computational grid formed by two non-dedicated Beowulf clusters located across the University of Edinburgh, configured as shown in Table 2. The grid fabric, which includes

<table>
<thead>
<tr>
<th></th>
<th>\texttt{bw240}</th>
<th>\texttt{bw530}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{Hardware}</td>
<td>64 nodes</td>
<td>16 nodes</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel P4</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td>Memory</td>
<td>1 GB / node</td>
<td>2 GB / node</td>
</tr>
<tr>
<td>Network</td>
<td>2x100Mb/s</td>
<td>1x100Mb/s, 1x1Gb/s Myrinet</td>
</tr>
<tr>
<td>BogoMips</td>
<td>3350–3555</td>
<td>3326–3359</td>
</tr>
</tbody>
</table>

Table 2. The Beowulf Clusters \texttt{bw240} & \texttt{bw530}

Hardware Configuration
a non-dedicated network, a storage sub-system and individual cluster management software, enables the interconnection and storage virtualisation while maintaining both clusters as two separate entities. We consider this grid environment representative as it spans different administrative domains, comprises heterogeneous nodes and links, and does not have dedicated resource co-allocation or reservation.

This environment is aligned with Foster’s checklist for a grid [9], where he points out that a grid must coordinate resources as opposed to centralise them; use open standards and interfaces; and, most important to this work, deliver non-trivial qualities of service. A similar industry-oriented perspective has been provided by Gentzsch [12] when he defines cluster grids as an interconnected set of local clusters, mainly oriented to heavy throughput computing. A similar position has recently been expressed in a series of personal interviews conducted by Stockinger [26]. As presented in table 2, our evaluation cluster grid environment adopts this conceptualisation.

In terms of software, all nodes were configured with Linux Red Hat FC5 with kernel 2.6, gcc 4.1.1 and LAM/MPI 7.1.2. All C programming modules were compiled with gcc using the “-pedantic -ansi -Wall -O2” flags.

For reproducibility purposes, we have employed as stage function the “whetstones” procedure from the 1997 version [21] of the Whetstone benchmark with parameters (256, 100, 0). It accounts for some 5 seconds of double-precision floating-point processing in an unloaded node of our experimental cluster grid environment.

All variability in the system is due to external load and, to a lesser extent, to the difference in performance among processors.

5.2 Tests

We have assembled an empirical evaluation using three node pools, \( n = \{8, 16, 32\} \), handling five input data sizes \( S = \{32, 64, 128, 256, 512\} \). The fifteen scenarios were staged on three different days, based on the number of nodes, with each scenario yielding to the execution of five experiments. To make the execution of each of the twenty-five experiments independent in a day, we interleaved their execution according to the input data size and ensured that just one pipeline experiment was executed at a time.

Each entry in table 3 presents the average of three experiments along with its coefficient of variation (cv), the average threshold for the series, and the naïve time estimate.

The obtained figures attest the performance improvements attributed to the use of our pipeline. Figure 2 presents a summary of the overall performance gain due to its use. Our empirical evaluation demonstrates a correlation between the size of the data input, \( S \), and performance gains, as load variations impact more as the processing increases.

While the estimate presumes an execution of the pipeline under steady conditions and neglects any load variation including any flurries, it is a guideline for quantifying the performance improvements.

In the course of the 3-day empirical evaluation, our grid entertained different workloads from a multiplicity of users. Such workloads provided a realistic environment to assert the adaptivity of our pipeline. To monitor the overall system load, we recorded the the Linux \texttt{uptime} command output (1-minute reading) during the entire duration of all the experiments. Figure 3 shows the results. Each row presents two views from an evaluation day.

Left Column A 3-D graph with each node load during the entire day. The x-axis specifies the time of the day and the y-axis the grid nodes. The different load readings are plotted in the z-axis using different line styles.

Right Column A summary chart with the average load of each node and of the whole system. The x-axis designates the node and the dashed steps in the y-axis indicates the average load per node. The solid line in the middle indicates the average of the overall system load.

The load average, \( \mu \), and the load standard deviation, \( \sigma \), are written in the bottom-right corner of charts (b), (d), and (f) of Figure 3. It is self-evident that the highest load variation was detected with 32 nodes (\( \sigma = 0.78 \)) and the lowest with 16 (\( \sigma = 0.19 \)).

In order to illustrate the responsiveness of the pipeline to the load variation, we can examine a re-calibration triggered by any given node during execution.

Figure 4 presents the load pattern for a certain node. Part (a) presents its pattern for the entire day, highlighting the region where the re-calibration was triggered, and part (b) shows an amplified view of that region (18:30–19:00). Note that in this case, the prompt re-calibration avoid a costly bottleneck, as the node load abruptly increased fourfold during the execution of the pipeline.

6. Related Work

While the general pipeline mapping problem is known to be intractable for a representative number of processors, different approaches have been conceived to address constrained cases. Due to the importance of this construct, several of the skeletal and pattern frameworks include a pipeline construct. In particular, we would like to draw our attention to the following:

- La Laguna Pipeline (llp) [14] furnishes a conceptual tool for static multi-stage allocation using algorithmic skeletons. By approaching the problem with a 0/1 knapsack problem method, llp is employed to develop
Figure 2. Summary of the Overall Improvements due to the Use of the Parallel Pipeline

Table 3. Adaptive Pipeline Execution Times ($n = \{8, 16, 32\}, S = \{32, 64, 128, 256, 512\}$)
Figure 3. Complete Load Patterns for the 3-day Evaluation
a theoretical solution to stage scheduling. Interesting empirical results have been reported for heterogeneous systems using this analytical modeler in multi-cluster configurations with short-span tasks [1]

• Using process algebra to enable performance modelling, the eSkel [3] and the ASSIST [8] libraries have been augmented with performance-based mappings for distributed systems. In particular, eSkel has been successfully applied to the reformulation of the parallel solution of a problem in queueing theory [31]

• A series of heuristics have been developed for a simple pipeline skeleton executing on homogeneous and heterogeneous systems [4]

• PLPP [23] and CO₂P₃S [22] furnish generic parallel pipeline patterns as part of their programming frameworks

• Triana [13] and ICENI [11] deploy pattern-based programming frameworks which employ the pipeline pattern and its derivatives to deploy scientific workflows

Nonetheless, the main difference between our GRASP–based pipeline and the aforementioned approaches is that it is intended to be adaptable by construct, and focused on empirical, system-infrastructure methodologies. Our pipeline can forecast and enhance the performance of a skeletal application in a grid by exploiting the knowledge of the skeletal structure while efficiently preserving the skeletal behaviour.

7. Concluding Remarks

The adaptive approach has responded well under varying load conditions, since the execution times in the non-adaptive parametrisation have increased at a considerably higher rate than the adaptive ones.

With respect to the analysis of the mapping problem, the findings of this paper provide an alternate approach, using an application-oriented calibration in order to forecast grid resource utilisation. This tacitly reinforces the notion that although computational grids are highly dynamic, forecasts based on historical resource utilisation can accurately provide guidance to distribute workloads.

From a performance standpoint, it is arguable that the overall performance of our pipeline pattern improves as long as variations in the bottleneck stage are controlled through continuous performance monitoring and process migration. Note that our pipeline reacted promptly to the load variations, preventing the development of a bottleneck for the pipeline processing. This is particularly useful as workload flurries —repetitive activity surges, caused by a single user, which dominate the workload for a relatively short period— have been traced as a cause of performance disruption, and many performance models filter them out [28]. However, it is a fact they exist in real non-dedicated systems and have to be dealt with in an efficient manner.

Acknowledgements

This work has been partly supported by CONACyT–Mexico under the doctoral scholarship number 81887.
References


