Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3–6, 1999 MIT, Boston, USA

Stream parallel skeleton optimization

M. Aldinucci & M. Danelutto
Department of Computer Science – University of Pisa – Italy
Email: {aldinuc,marcod}@di.unipi.it

Abstract We discuss the properties of the composition of stream parallel skeletons such as pipelines and farms. By looking at the ideal performance figures assumed to hold for these skeletons, we show that any stream parallel skeleton composition can always be rewritten into an equivalent "normal form" skeleton composition, delivering a service time which is equal or even better to the service time of the original skeleton composition, and achieving a better utilization of the processors used. The normal form is defined as a single farm built around a sequential worker code. Experimental results are discussed that validate this normal form.
Keywords: skeletons, rewriting, stream parallelism.

1 Introduction

Skeleton based programming models represent an interesting subject in the field of parallel programming models. Cole introduced the skeleton concept in the late 80's [1]. Cole's skeletons represented parallelism exploitation patterns that can be used (instantiated) to model common parallel applications. Later, the skeleton concept evolved. Different authors recognized that skeletons can be used as constructs of an explicitly parallel programming language, actually as the exclusive way in these languages to express parallel computations [2, 3, 4]. Recently, the skeleton concept evolved again, and became the coordination layer of structured parallel programming environments [5, 6, 7]. In all cases, however, a skeleton can be understood as a higher order function taking one or more other skeletons or portions of sequential code as parameters, and modeling a parallel computation out of them. Different parameters passed to the same skeleton give different parallel applications, all exploiting the same kind of parallelism: the one encapsulated by the skeleton used.

In most cases, in order to implement skeletons efficiently on parallel architectures, compiling tools based on the concept of the implementation template have been developed [1, 8, 9]. The implementation templates are known, efficient and parametric ways (process networks, actually) of implementing a skeleton on a given target architecture. Therefore, the process of generating parallel code from a skeleton source code can be performed by looking at the skeleton nesting in the source code and deriving a corresponding nesting of (suitably instantiated) implementation templates [10, 9]. Furthermore, due to the fact that the skeletons have a clear functional and parallel semantics, different rewriting techniques have been developed (or derived from other functional programming contexts [11, 12]) that allow skeleton programs to be transformed/rewritten into equivalent ones, achieving different performance when implemented on the target architecture [2]. These rewritings can also be driven by some kind of analytical performance models associated with the implementation templates of the skeletons, in such a way that only those rewritings leading to more efficient implementation of the skeleton code at hand are considered [13].

The skeletons considered by the structured parallel programming community range from a small number of very simple, general purpose skeletons [1, 2, 8] to a large number of very specific, application oriented skeletons [14]. Usually, the skeleton set comprises both data parallel and control parallel skeletons. The data parallel skeletons model parallel computations whose parallel activities come from the computation of a single data item, whereas control parallel skeletons model parallel computations whose parallel activities come from the computation of different, independent data items. In turn, stream parallel skeletons are those control parallel skeletons exploiting parallelism in the computation of streams of results out of streams of (independent) data items.

In this paper, we discuss the properties of programs derived by nesting stream parallel skeletons and implemented via implementation templates. Both the skeletons and the implementation templates considered are those used in P3L and SklEcl [8, 15]. In particular, we will show how arbitrary compositions of stream parallel skeletons, including pipelines and farms, can always be rewritten as farms of sequential code and we will prove that this second form delivers a better service time.

[0]This work has been partially supported by the MOSAICO and PQE2000 italian projects.

2 The skeleton framework

We consider a skeleton framework containing only *stream parallel* skeletons, i.e. we do not take into account data parallel skeletons. In particular, we consider skeletons modeling pipeline and farm parallelism as well as sequential composition of sequential computations, similar to the ones used in [2, 8].

Stream parallel skeletons exploit parallelism in the computation of streams of results from streams of input data. By data stream we mean an ordered, finite collection of homogeneous (i.e., having the same type) data items, possibly being available at different times. We denote a data stream with data items $x_1, \ldots, x_n$ by $\langle\ x_n, \ldots, x_1\ \rangle$ and we assume that data item x_i is available immediately after data item x_{i-1} was available and immediately before x_{x+1} is be available. When computing any stream parallel skeleton σ onto a data stream $\langle x_n, \ldots, x_1\rangle$, the computation of any result data item appearing onto the output stream $\langle y_n, \ldots, y_1\rangle$ is independent of the computation of any other result data item. In other words, according to the usual definition of pipeline and farm skeletons (such as the one given by [2, 8]), our stream parallel skeletons denote *stateless* computations. Therefore, given any skeleton program σ an input data stream $\langle\ x_n, \ldots, x_1\ \rangle$ and the corresponding output data stream $\langle\ y_n, \ldots, y_1\ \rangle$, it holds that $\forall i \in [1, n]\ y_i = \mathcal{F}(x_i)$ provided that $\mathcal{F}(x_i)$ is the function computed by skeleton σ.

In the following paragraphs, we give the functional and parallel semantics of the skeletons included in our set. The functional semantics (denoted in the following by $\mathcal{F}$) just denotes the results computed by each skeleton, and it will be used to show that different skeleton programs (i.e. programs exploiting different skeleton nestings) compute the same results. The functional semantics of our skeletons will be given by formally defining the function $\mathcal{F}$. The (informal) parallel semantics, instead, denotes the parallelism exploitation patterns of the skeletons and will be used to derive the analytical cost models of Section 2.2, that, in turn, will be used to prove the effectiveness of normal form, stream parallel skeleton programs discussed in Section 3. For each skeleton, the informal parallel semantics will be given by discussing the parallelism exploitation pattern used to implement the skeleton. Of course, both functional and parallel semantics of a skeleton must be taken into account in order to understand the skeleton peculiarities.

Sequential skeleton The sequential skeleton (denoted by the keyword `seq`) simply transforms a sequential portion of code, written in some language l in a skeleton that can be used as a parameter of other skeletons, such as the pipeline, the farm and the sequential composition ones. We assume that a function $\mathcal{H}_l$ exists, for any sequential language l, such that $\mathcal{H}_l[\mathtt{prog}] = \phi : \alpha \to \beta$[1], where ϕ is the function computed by the sequential fragment of code `prog`, the type of input data processed by ϕ is α and the type of output data produced by ϕ is β. Therefore, the function computed by a sequential skeleton is (its functional semantics is defined by): $\mathcal{F}[\mathtt{seq}(\mathtt{prog})] = \mathcal{H}_l[\mathtt{prog}] = \phi$ No parallelism is exploited while computing a sequential skeleton, i.e. we assume that the whole computation of $\mathcal{F}[\mathtt{seq}(\mathtt{prog})]$ is performed sequentially on a single processing element.

Sequential composition The sequential composition skeleton (denoted by the infix operator ";") models the sequential composition of sequential skeletons. Therefore, the function computed by the skeleton: $\iota_1; \ldots; \iota_k$ turns out to be[2]: $\mathcal{F}[\iota_1; \ldots; \iota_k] = \mathcal{F}[\iota_k] \circ \ldots \circ \mathcal{F}[\iota_1]$ provided that each $\iota_i = \mathtt{seq}(\mathtt{prog})$[3], and the type of the resulting function is $\alpha_1 \to \alpha_{k+1}$ (provided that $\forall i\ \mathcal{F}(\iota_i) : \alpha_i \to \alpha_{i+1}$). No parallelism is exploited when evaluating the sequential composition skeleton, i.e. we assume that the whole computation of $\mathcal{F}[\iota_1; \ldots; \iota_k]$ is performed on a single processing element.

Pipeline The pipeline skeleton (denoted by the infix operator "|") models function composition. Whenever a function f of the input data has to be computed, such that the function can be expressed as the composition of some functions $f_1, \ldots, f_k$, i.e. $f = f_k \circ \ldots \circ f_1$, a pipeline skeleton can be used. Given a pipeline skeleton, such as $\sigma_1 | \ldots | \sigma_k$ the function computed by the skeleton is $\mathcal{F}[\sigma_1 | \ldots | \sigma_k] = \mathcal{F}[\sigma_k] \circ \ldots \circ \mathcal{F}[\sigma_1]$ and the type of the resulting function is $\alpha_1 \to \alpha_{k+1}$ (provided that $\forall i\ \mathcal{F}(\sigma_i) : \alpha_i \to \alpha_{i+1}$). Parallelism is exploited in the computation of a pipeline skeleton as the computations relative to the different stages on different data items can be performed in parallel. In principle, given a k stage pipeline operating on an input data stream $\langle x_n, \ldots, x_1\rangle$, the computation of stage i (i.e. the computation of $\mathcal{F}[\sigma_i]$) relative to the input data item x_j, can be performed in parallel with the computations of stages $i' \in (1, k)$ s.t. $i' \neq i$ relative to the input data items $x_{j-(i'-i)}$.

Farm The farm skeleton (denoted by the prefix operator "$\diamond$") models functional replication. Given a farm skeleton $\diamond(\sigma)$ the function computed is given by $\mathcal{F}[\diamond(\sigma)] = \mathcal{F}[\sigma]$ and the type of the resulting function will be $\mathcal{F}[\diamond(\sigma)] : \alpha \to \beta$ provided that $\mathcal{F}[\sigma] : \alpha \to \beta$. The farm skeleton has to be interpreted as identity, from the strict viewpoint of its functional semantics. Its effectiveness derives from its parallel semantics: in the computation of a farm skeleton, parallelism is exploited as the computations relative to different (available) items of the input data stream can all be per-

[1] $f : \alpha \to \beta$ denotes that function f has type $\alpha \to \beta$.

[2] $\circ$ denotes the usual function composition $(g \circ f)(x) = g(f(x))$.

[3] In the following, we will denote by ι_i sequential skeletons such as `seq(prog)`.

formed in parallel. Therefore given a farm skeleton such as $\diamond(\sigma)$ and an input stream such as $\langle x_n, \ldots, x_1\rangle$ the computations $\mathcal{F}[\sigma](x_i)$ and $\mathcal{F}[\sigma](x_j)$ $(i \neq j)$ can be performed in parallel once the data items x_i and x_j are available on the input stream.

Using the skeleton framework described above, skeleton expressions (i.e. programs) such as the following may be written:

```
Threshold = seq ...⟨C code here⟩... endseq
Contour = seq ...⟨C code here⟩... endseq
Recognize = seq ...⟨C code here⟩... endseq
◇(Threshold | Contour | Recognize)
```

Assuming that the C code in sequential skeleton `Threshold` processes `Bitmaps` in order to keep black the pixels whose color value is above a given threshold ($\mathcal{F}$[`Threshold`] : `Bitmap` $\rightarrow$ `Bitmap`), that the C code in the `Contour` skeleton processes black and white bitmaps in order to recognize contour lines ($\mathcal{F}$[`Contour`] : `Bitmap` $\rightarrow$ `Contour`[]) and, eventually, that the C code in `Recognize` recognizes printable characters out of a set of contour lines ($\mathcal{F}$[`Recognize`] : `Contour`[] $\rightarrow$ `char`[]), the skeleton expression $\diamond$(`Threshold` | `Contour` | `Recognize`) for each `Bitmap` of an input data stream places onto an output data stream a `char` vector holding all the characters recognized in the `Bitmap`. Parallelism is exploited both by overlapping the computations of `Threshold`, `Contour` and `Recognize` relative to one of the `Bitmaps` appearing onto the input stream (this is because of the pipeline skeleton) *and* by performing in parallel the computations relative to the different `Bitmaps` appearing onto the input stream (this is because of the outermost farm skeleton).

2.1 Stream parallel rewriting rules

Many equivalences can be established between skeleton expressions involving the stream parallel skeletons defined in Section 2. Those equivalences can be proved by looking at the functional semantics of the skeletons involved. As an example, given the two skeleton expressions σ and $\diamond(\sigma)$, we can immediately see that both compute the same function $\mathcal{F}(\sigma)$, due to the fact that farm represents identity from the functional semantics viewpoint. Therefore we can conclude that, from the point of view of the results computed by the programs denoted by the two expressions, $\sigma \equiv \diamond(\sigma)$. This, in turn, leads to the two rewriting rules **Fe** (farm elimination) and **Fi** (farm introduction) of Figure 1: the **Fi** rule allows farms to be introduced on top of any skeleton expression, whereas **Fe** rule allows farms to be removed from the top of any skeleton expression. Both rules preserve the functional semantics. Instead, parallel semantics is not preserved, since different kinds and, consequently, different amounts of parallelism are exploited in the left-hand and right-hand side.

$\sigma \rightarrow \diamond(\sigma)$	**(Fi)**
$\diamond(\sigma) \rightarrow \sigma$	**(Fe)**
$(\sigma_1\|(\sigma_2\|\sigma_3)) \rightarrow ((\sigma_1\|\sigma_2)\|\sigma_3)$	**(Pas1)**
$((\sigma_1\|\sigma_2)\|\sigma_3) \rightarrow (\sigma_1\|(\sigma_2\|\sigma_3))$	**(Pas2)**
$(\iota_1;(\iota_2;\iota_3)) \rightarrow ((\iota_1;\iota_2);\iota_3)$	**(SCas1)**
$((\iota_1;\iota_2);\iota_3) \rightarrow (\iota_1;(\iota_2;\iota_3))$	**(SCas2)**
$;(\iota) \rightarrow \iota$	**(Se)**
$\iota \rightarrow ;(\iota)$	**(Si)**
$(\iota_1\|\ldots\|\iota_k) \rightarrow (\iota_1;\ldots;\iota_k)$	**(Coll)**
$(\iota_1;\ldots;\iota_k) \rightarrow (\iota_1\|\ldots\|\iota_k)$	**(Expd)**

Figure 1: *Rewriting rules*

Figure 1 presents some of the rewriting rules we can prove for the skeleton set taken into account in this paper (the acronyms on the right stand for farm introduction and elimination (**Fi**, **Fe**), pipeline and sequential composition associativity (**Pas1**, **Pas2**, **SCas1**, **SCas2**), sequential composition elimination and introduction (**Se**, **Si**), pipe collapse and sequential composition expansion (**Coll**, **Expd**)). These rules state that farms can be introduced or removed from skeleton expressions, that pipeline and sequential composition skeletons are associative, that the sequential composition of a single sequential skeleton is equivalent to the sequential skeleton itself, and that pipeline and sequential skeletons can be freely interchanged. Due to pipeline and sequential composition associativeness, we will simplify in the following $(\ldots((\iota_1;\iota_2);\iota_3);\ldots\iota_n)$ by $\iota_1;\ldots;\iota_n$.

These transformations can be performed without affecting the result computed by the program, but affecting the amount and the kind of parallelism exploited and, therefore, the performance eventually achieved. Despite their simplicity (we do not prove their validity here, but the proofs are straightforward), these rules can be effectively used to rewrite any stream parallel skeleton expression into a "normal form" expression (defined in Section 3) which will be proven to be more efficient that the non-normal, original one.

2.2 Performance estimation

We assume that programs written with the skeletons of Section 2 are compiled to parallel object code using template based compiling tools, such as those used in both P3L and SklEcl [8, 15, 9, 7]. Such kind of tools are based on the existence of a predefined, parametric and efficient process network for each one of the skeletons. These parametric process networks, the implementation templates, are used to implement the skeleton instances. Here, in particular, we assume that each implementation template is a parametric (in the number of processing nodes used) process network with a single input and a single output "point". That is, we assume that there is a unique place where data of the input stream are taken and another unique place where data of the output stream are placed by the template process network (these "places" can be channels, memory locations, etc.). Although this assumption may look like to be restrictive, it perfectly models the concept of data stream, which is a *single* entity hosting data items available at different, consecutive times. Furthermore, this assumption allows effective implementation template composition to be performed, which in turn is necessary to implement full skeleton nesting [9]. This is also the assumption currently made in the compiling tools of both P3L and SklEcl, the skeleton based parallel programming environments currently being developed in Pisa.

In order to estimate the performance of our skeleton programs, we need some performance model associated to the implementation templates we assume to use to implement the skeletons on the target architecture. Such performance models should reflect the performance achieved when implementing a particular process network on a particular target architecture, however, and therefore they are strongly related to the templates used and to the target machine used. Instead, we are interested in the general properties of the skeletons, and therefore here we try to abstract the performance models of the templates from templates peculiarities and from target architecture features. Eventually, we come up with "ideal" performance models that have to be intended as asymptotic models, i.e. models that represent the *best* performance that can be achieved by a template implementing the skeleton under the assumptions made in this work. This because we are interested in the evaluation of the lower bounds, rather than in the precise modeling of the template performance.

In this work we consider the service time of a template as the performance measure to be optimized. The service time is the time occurring between the delivery of two distinct, consecutive result data items onto the output stream. It can also be defined as the time necessary to the first process of the template to accept and process a new input data item [4].

We introduce now the ideal performance models for the implementation templates of our skeletons. We denote by $T_s(\sigma)$ the service time of the template implementing the skeleton σ, and by $T_{seq}(\iota)$ the average amount of time spent in computing the code embedded by the seq skeleton $\iota = \mathsf{seq}(\ldots)$. We denote by $T_i(\sigma)$ and $T_o(\sigma)$ the time spent by the template implementing σ in receiving an input data item and delivering an output data item from/to a distinct processing node. These parameters can be used to model either the overhead associated with a communication or the actual (total) time spent in the communication, depending on the parallelism degree of the target architecture processing element (i.e. on the presence of some kind of independent communication processor).

Sequential skeleton: $T_s(\iota) = T_i(\iota) + T_o(\iota) + T_{seq}(\iota)$ i.e. the service time of a sequential skeleton should be at least equal to the time spent in receiving the input data plus the time spent in executing the code, plus the time spent in delivering the result data.

Sequential composition skeleton: $T_s(\iota_1;\ldots;\iota_k) = T_i(\iota_1)+T_o(\iota_k)+\sum_{i=1}^{k} T_{seq}(\iota_i)$ i.e. the service time of a sequential composition of sequential skeletons should be at least equal to the time spent in gathering parameters and delivering results plus the time spent in the computation of each one of the seq skeletons involved.

Pipeline: $T_s(\sigma_1|\ldots|\sigma_k) = \uparrow\{T_s(\sigma_1),\ldots,T_s(\sigma_k)\}$[5] i.e. the service time of a pipeline should be at least equal to the maximum of the service times of its stages [16].

Farm: $T_s(\diamond\sigma) = \downarrow\{\uparrow\{T_i(\sigma), T_o(\sigma)\}, T_s(\sigma)\}$, denoting the fact that a farm template with single input/output points cannot serve requests in a time shorter than the time spent in receiving an input or delivering an output data item. Furthermore, it does not make sense to use a farm when the time spent in delivering a data item is greater that the time spent in computing the result of that data item, as in this case the results could be computed with better service time without using the farm [17]. It's worthwhile pointing out that this kind of performance modeling of the farm corresponds to the assumption that a farm template is basically a three stage pipeline: the first stage gathers data items from the input data stream, the second (parallel!) stage computes the farm results and the third stage gathers the results from the second one and delivers them on the output stream.

3 Collapsing skeletons

We first define a "normal form" of stream parallel skeleton compositions, then we will show how normal forms of skeleton compositions always achieve an equal or better performance (in terms of the service time) w.r.t. the equivalent, non-normal form skeleton compositions, despite their simpler structure in terms of the skeleton nesting used.

Given any skeleton composition Δ, we define $\mathsf{fringe}(\Delta)$ to be the ordered list of all the sequential portions of code in Δ. Formally, we define $\mathsf{fringe}(\Delta)$ by induction on the structure of Δ as follows:

- $\mathsf{fringe}(\iota) = \iota$
- $\mathsf{fringe}(\iota_1;\ldots;\iota_k) = [\iota_1,\ldots,\iota_k]$
- $\mathsf{fringe}(\diamond(\sigma)) = \mathsf{fringe}(\sigma)$
- $\mathsf{fringe}(\sigma_1|\sigma_2) = \mathsf{append}(\mathsf{fringe}(\sigma_1), \mathsf{fringe}(\sigma_2))$

[4]This measure is different from the completion time, i.e. the time occurring between the arrival of the first data item of the input stream to the template and the delivering of the last result data item onto the output stream.

[5]We denote by $\uparrow\{\ldots\}$ and $\downarrow\{\ldots\}$ the maximum and the minimum of a set, respectively.

We define the **normal form** of a generic skeleton composition Δ, denoted by $\overline{\Delta}$, as[6]

$$\overline{\Delta} = \diamond(;(\mathsf{fringe}(\Delta)))$$

It is easy to prove the following statement:

Statement 1 *Given any skeleton composition* Δ, $\mathcal{F}[\Delta] = \mathcal{F}[\overline{\Delta}]$

Proof The proof is by induction on the structure of the skeleton program.
Base case The program just contains a sequential skeleton or a sequential composition one. In this case either $\Delta = \iota = \texttt{seq(...)}$ and therefore $\iota \overset{Si}{\to} ;(\iota) \overset{Fi}{\to} \diamond(;(\iota))$ or $\Delta = \iota_1; \ldots ;\iota_k$ and therefore $\iota_1; \ldots ;\iota_k \overset{Fi}{\to} \diamond(\iota_1; \ldots ;\iota_k)$
Inductive cases The program is either a pipe or a farm of normal form skeletons. In this case

- either $\Delta = (\diamond(\iota_{11}; \ldots ;\iota_{1n_1}))| \ldots |(\diamond(\iota_{k1}; \ldots ;\iota_{kn_k}))$, therefore
  $(\diamond(\iota_{11}; \ldots ;\iota_{1n_1}))| \ldots |(\diamond(\iota_{k1}; \ldots ;\iota_{kn_k})) \overset{Fe}{\to}$
  $(\iota_{11}; \ldots ;\iota_{1n_1})| \ldots |(\iota_{k1}; \ldots ;\iota_{kn_k}) \overset{Coll}{\to}$
  $(\iota_{11}; \ldots ;\iota_{1n_1}); \ldots ;(\iota_{k1}; \ldots ;\iota_{kn_k}) \overset{SCas}{\to}$
  $(\iota_{11}; \ldots ;\iota_{1n_1}; \ldots ; \iota_{k1}; \ldots ;\iota_{kn_k}) \overset{Fi}{\to}$
  $\diamond(\iota_{11}; \ldots ;\iota_{1n_1}; \ldots ;\iota_{k1}; \ldots ;\iota_{kn_k})$,
- or $\Delta = \diamond(\diamond(\iota_1; \ldots ;\iota_n))$ and therefore
  $\diamond(\diamond(\iota_1; \ldots ;\iota_n)) \overset{Fe}{\to} \diamond(\iota_1; \ldots ;\iota_n)$

3.1 Collapsing effects

Using the abstract performance models for templates introduced in Section 2.2, we can prove the following result, which is the main result discussed in this paper.

Statement 2 *Given any stream parallel composition* Δ*, such that* $\forall \iota \in \mathsf{fringe}(\Delta)\ T_i(\iota) < T_{seq}(\iota)$ *and* $T_o(\iota) < T_{seq}(\iota)$*, then* $T_s(\overline{\Delta}) \leq T_s(\Delta)$

Proof We prove the statement by induction on the structure of Δ.
Base cases Either $\Delta = \iota$ or $\Delta = \iota_1; \ldots ;\iota_k$ and therefore:

- $\Delta = \iota$
  $T_s(\Delta) = \downarrow\{T_s(\Delta)\} \geq \downarrow\{\uparrow\{T_i(\iota), T_o(\iota)\}, T_s(\iota)\} = T_s(\diamond(;(\iota))) = T_s(\overline{\Delta})$
- $\Delta = \iota_1; \ldots ;\iota_k$
  $T_s(\Delta) = \downarrow\{T_s(\Delta)\} \geq$
  $\downarrow\{\uparrow\{T_i(\iota_1), T_o(\iota_k)\}, T_s(\Delta)\} =$
  $\downarrow\{\uparrow\{T_i(\iota_1), T_o(\iota_k)\}, T_s(\iota_1, \ldots, \iota_k)\} =$
  $T_s(\diamond(\iota_1, \ldots, \iota_k)) = T_s(\overline{\Delta})$

Inductive cases Either $\Delta = \diamond\sigma$ or $\Delta = \sigma_1|\sigma_2$ and therefore:

- $\Delta = \diamond\sigma$. In this case $T_s(\sigma) \geq T_s(\overline{\sigma})$ by induction hypothesis. Furthermore, we assume that $\mathsf{fringe}(\sigma) = \iota_1, \ldots, \iota_k$ and therefore:
  $T_s(\Delta) = T_s(\diamond\sigma) = \downarrow\{\uparrow\{T_i(\sigma), T_o(\sigma)\}, T_s(\sigma)\} \geq$
  $\downarrow\{\uparrow\{T_i(\sigma), T_o(\sigma)\}, T_s(\overline{\sigma})\} =$
  $\downarrow\{\uparrow\{T_i(\iota_1), T_o(\iota_k)\}, T_s(\overline{\sigma})\} =$
  $\downarrow\{\uparrow\{T_i(\iota_1), T_o(\iota_k)\}, \downarrow\{\uparrow\{T_i(\iota_1),$
  $T_o(\iota_k)\}, T_s(\iota_1; \ldots ;\iota_k)\}\} =$
  $\downarrow\{\uparrow\{T_i(\iota_1), T_o(\iota_k)\}, \uparrow\{T_i(\iota_1), T_o(\iota_k)\},$
  $T_s(\iota_1; \ldots ;\iota_k)\} =$
  $\downarrow\{\uparrow\{T_i(\iota_1), T_o(\iota_k)\}, T_s(\iota_1; \ldots ;\iota_k)\} =$
  $T_s(\diamond(\iota_1; \ldots ;\iota_k)) = T_s(\overline{\diamond\sigma})$
- $\Delta = \sigma_1|\sigma_2$ (with $\mathsf{fringe}(\sigma_1) = \iota_{11}, \ldots, \iota_{1n_1}$ and $\mathsf{fringe}(\sigma_2) = \iota_{21}, \ldots, \iota_{2n_2}$). In this case, $T_s(\sigma_i) \geq T_s(\overline{\sigma_i}), i \in \{1, 2\}$ by induction hypothesis:
  $T_s(\sigma_1|\sigma_2) = \uparrow\{T_s(\sigma_1), T_s(\sigma_2)\} \geq$
  $\uparrow\{T_s(\overline{\sigma_1}), T_s(\overline{\sigma_2})\} =$
  $\uparrow\{T_s(\diamond(\iota_{11}; \ldots ; \iota_{1n_1})), T_s(\diamond(\iota_{21}; \ldots ; \iota_{2n_2}))\} =$
  $\uparrow\{\downarrow\{\uparrow\{T_i(\iota_{11}), T_o(\iota_{1n_1})\}, \sum_{i=11}^{1n_1} T_{seq}(\iota_i)\},$
  $\downarrow\{\uparrow\{T_i(\iota_{21}), T_o(\iota_{2n_2})\}, \sum_{i=21}^{2n_2} T_{seq}(\iota_i)\}\} =$
  $\uparrow\{\uparrow\{T_i(\iota_{11}), T_o(\iota_{1n_1})\}, \uparrow\{T_i(\iota_{21}), T_o(\iota_{2n_2})\}\} =$
  $\uparrow\{T_i(\iota_{11}), T_o(\iota_{1n_1}), T_o(\iota_{2n_2})\} \geq$
  $\uparrow\{T_i(\iota_{11}), T_o(\iota_{2n_2})\} \geq$
  $\downarrow\{\uparrow\{T_i(\iota_{11}), T_o(\iota_{2n_2})\}\} \geq$
  $\downarrow\{\uparrow\{T_i(\iota_{11}), T_o(\iota_{2n_2})\}, \sum_{i=11}^{1n_1} T_{seq}(\iota_i) +$
  $\sum_{i=21}^{2n_2} T_{seq}(\iota_i)\} =$
  $T_s(\diamond(\iota_{11}; \ldots ;\iota_{2n_2})) = T_s(\overline{\sigma_1|\sigma_2})$

This result tells us that any time we have a stream parallel composition in a larger skeleton composition we can rewrite it in normal form and expect that performance (service time) is either the same or better than the original one. The assumptions on the input and output times ($\forall \iota \in \mathsf{fringe}(\Delta)\ T_i(\iota) < T_{seq}(\iota)$ and $T_o(\iota) < T_{seq}(\iota)$) are not restrictive in that every time we have a module whose latency is smaller than the time spent to feed the module with new data to be computed and to extract results out of the module, the parallel module can be conveniently eliminated and a sequential module can be used instead.

Concerning the effects of collapsing stream parallel skeleton compositions to normal form on the resources used, a different kind of reasoning has to be performed. The farm introduced by the normal form compute "heavy" functions, the functions obtained by merging all the sequential stages of the original composition. Therefore, in order to be effective, a large number of parallel processes have to be included in the farm template, in such a way that, every time a new data item is available to schedule a new parallel computation, a process happens to be idle, ready to start this new computation. This, in turn, allows to achieve the asymptotic performance modeled by performance formulas of Section 2.2. The number of parallel processes used in the normal form farm should therefore be something like $\frac{T_s(\iota_1; \ldots ; \iota_k)}{\uparrow\{T_i(\iota_1), T_o(\iota_k)\}}$. Whether or

[6] We use the sequential composition operator in prefix form, in this case, assuming that $;(\iota_1, \ldots, \iota_k) \equiv \iota_1; \ldots ;\iota_k$.

TableA	$\iota_1;\iota_2$	$\diamond(\iota_1;\iota_2)$	$\diamond(\diamond(\iota_1)\mid\diamond(\iota_2))$	$(\diamond(\iota_1)\mid\diamond(\iota_2))$	$\diamond(\iota_1\mid\iota_2)$	$\diamond(\iota_1)\mid\iota_2$	$\iota_1\mid\diamond(\iota_2)$
T_s	6.03	0.33	0.35	0.37	0.35	1.08	4.98
T_c	1207.76	71.11	76.60	81.00	74.64	222.04	1003.75
#PE	1	24	44	24	34	9	7
ϵ (%)		75.60	38.85	66.99	50.71	62.05	17.29

TableB	$\iota_1;\iota_2$	$\diamond(\iota_1;\iota_2)$	$\diamond(\diamond(\iota_1)\mid\diamond(\iota_2))$	$(\diamond(\iota_1)\mid\diamond(\iota_2))$	$\diamond(\iota_1\mid\iota_2)$	$\diamond(\iota_1)\mid\iota_2$	$\iota_1\mid\diamond(\iota_2)$
T_s	6.03	0.39	1.30	0.72	0.43	1.12	4.99
T_c	1207.76	84.50	286.62	151.67	91.43	230.35	1004.69
#PE	1	20	20	20	20	20	20
ϵ (%)		75.52	23.08	41.93	69.56	26.88	6.04

Figure 2: *Normal form vs. non-normal forms.* ***Table A****: optimal number of processing elements for each run* ***Table B****: same number of processing elements for each run (T_s: service time, T_c: completion time, #PE: number of processing elements used, ϵ: efficiency)*

not this amount of processing resources (parallel processes) is greater than the amount of resources needed to implement the corresponding non-normal form is not known, at the moment, but we are currently looking for a result, similar to the one discussed above for the service time, that could assess the relationships between the resources used in the normal and non-normal forms of stream parallel skeletons compositions. We expect that normal form programs can be implemented with a smaller amount of resources dedicated to the skeleton run time support, because of the simpler skeleton structure of normal form programs with respect to non-normal form ones. This should lead to smaller overheads in the execution of normal forms and, in general, to a better utilization of the processing elements at hand.[7] Preliminary experimental results (shown in Section 3.2) seem to validate this kinds of reasoning.

3.2 Experimental results

We performed some experiments aimed at validating the theoretical results discussed in Section 3.1. We used two skeleton compilers, Anacleto [10], the Pisa prototype P3L compiler, and the skeleton compiler integrated within SkIE [6]. Both compilers handle a skeleton language including the stream parallel skeletons discussed in Section 2, and both use implementation templates such as those discussed in Section 2.2.

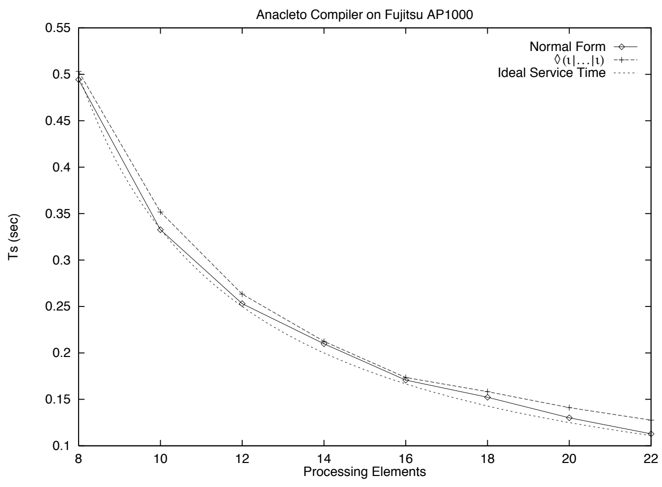
In order to check whether normal form programs perform better than equivalent, non-normal form ones, we performed multiple runs of normal and non-normal versions of the same programs on a cluster of 10 Pentium II PCs interconnected by a 100Mbit switched Ethernet and on a Fujitsu AP1000 parallel machine.

First of all, we looked at the behavior of the different, equivalent forms of a program, with respect to performance. Table A of Figure 2 summarizes the service and completion time, the resource usage count and the efficiency measured when running different forms (equivalent skeleton compositions) of the same program. In this case, the program is a two stage pipeline with the first stage taking five times the time taken by the second one to compute a result.[8] All the different versions compute a stream of 200 input tasks (on a Fujitsu AP1000). In this first set of runs, we used the exact amount of resources (processing elements) that maximizes the run performance. This amount of resources can be derived by the performance models associated to the implementation templates used to implement the skeletons and depends, of course, on the kind of skeleton nesting used within the program. The normal form (second column, in the table) run takes less time to complete (T_C), delivers a better service time (T_S) and presents a better efficiency (computed on the service time) than the others, semantically equivalent forms of the program. In particular, the normal form uses the same amount of processing elements of the pipe of farms version, but this version of the program is slower.

In Table B of Figure 2, we summarize the times taken when the different versions of the programs are run using the same amount of processing elements. The amount of processing elements used has been chosen to be slightly smaller of the "optimal" amount of processing elements required by the normal form and farm of pipeline programs. In this case, the advantage coming from the usage of the normal form is more evident. This is due to the "better" usage of the available processing elements by normal form (i.e. to the smaller overheads introduced in computations and to the smaller number of processing elements dedicated to the execution of the bare skeleton/template run time support).

[7]We expect to achieve better service times *and*, at the same time, better efficiency.

[8]The latencies of sequential computations are randomly chosen in accordance with a normal distribution $N(\mu,\sigma)$ with variance $\sigma = 0.6$.



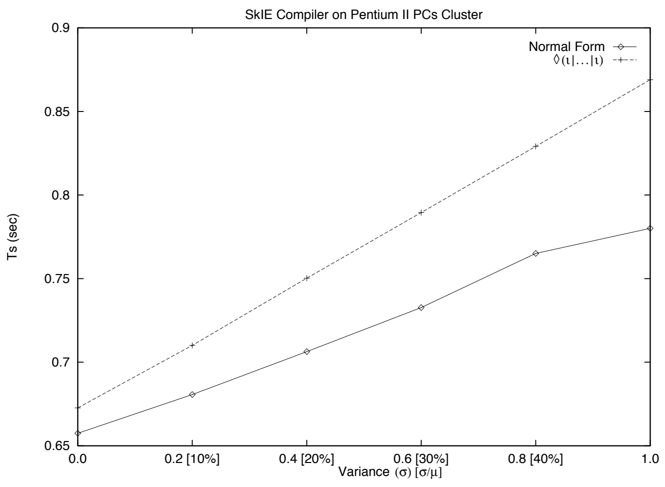


Figure 3: *Experimental results: service time (in seconds) vs. number of processing elements used (left) service times (in seconds) vs. variance of sequential skeleton time (right)*

We also considered programs that are expected to pay a minor performance slowdown with respect to the normal form programs: we considered skeleton compositions in the form $\diamond(\iota_1|\ldots|\iota_k)$ vs. the equivalent normal forms $\diamond(\iota_1;\ldots;\iota_k)$. In both cases the mean load of all sequential computations is equal and fixed. We expected minimum differences between the service times of first and second form of these programs, because of the poor structuring of the first form. Furthermore, we expected that normal form is not sensibly better than the other form when the difference between times spent in sequential computations is low, as in this case the load balancing effects of normal form farm are poorer. Figure 3 (left) plots the service time of a $\diamond(\iota_1|\ldots|\iota_k)$ program in function of the parallelism degree (the number of processing elements used, actually) vs. the service time of the equivalent normal form program (times taken on a Fujitsu AP 1000). The ideal service time (i.e. the service time computed by using the performance models of Section 2.2) of normal form program is also plotted. The normal form times are better than non-normal form times even if the load of all the sequential computations is exactly the same (and this should be the best condition for farms of pipelines with respect to the corresponding normal form: the farm of sequential composition skeleton) and, furthermore, normal form times are very close to the ideal ones.

In order to evaluate the effect of the local load imbalances on normal and non-normal form programs, we run programs with a variable variance in the sequential latencies. Figure 3 (rigth) plots the service times of programs when the latencies of sequential computations are randomly chosen in accordance with a normal distribution $N(\mu,\sigma)$ (times taken on the Linux Pentium II/Fast Ethernet cluster). Again, the normal form times are better than those achieved with the equivalent non-normal form programs. Moreover, due to the better load balancing capabilities of the normal form, the gap between the performances grows as the unbalancing of sequential latencies grows. We obviously expect that even better results can be measured in case of more structured programs, as in this case normal form leads to a sensible decrease in the run time support code that has to be executed to distribute data amongst the processing elements involved in the parallel execution of skeleton code.

4 Related work and conclusions

Rewriting techniques with some kind of associated cost calculus have been developed by different research groups working on skeletons, parallel functional programming and general structured parallel programming.

Papers have been published demonstrating the potential usage of a particular cost calculus in the evaluation of program transformations [18, 19]. Other researchers developed transformation techniques aimed at improving specific data parallel computations [12], mostly derived from the Bird-Merteens parallel functional programming framework [11]. However, most of these works just deal with data parallel computations and the associated transformations.

Both the Darlington group at the Imperial College and the authors' group working on P3L discussed transformation rules involving both stream (control) parallel and data parallel skeletons [2, 8, 13]. Even in these works, however, no idea of "best" or "normal form" skeleton composition such as the one discussed here has been presented.

The main result discussed in this paper is that any arbitrary composition of stream parallel skeletons can be rewritten into an equivalent "normal form" skeleton composition. Such normal form skeleton composition computes the same results computed by the original program, delivering a service time which is better or equal to the service time of the original program. This result has been derived taking into account ideal performance models, i.e. not taking into account all the overheads associated with the exploitation of nested skeleton programs, which are quite hard to measure within the abstract cost model we devised for the skeleton implementations. Therefore it has to be considered "optimistic" in the sense that normal forms are subject to a smaller amount of overhead with respect to highly nested, equivalent, non-normal forms.

Experimental results demonstrated that normal form programs deliver a better service time than the equivalent, non-normal form ones. All the experiments we performed on different machines, with different compiler and different programs showed that the normal form programs run faster than the equivalent, non-normal ones even in those cases where we expected the behavior to be quite close (e.g. farm of pipeline of balanced sequential stages). Furthermore, in those cases where the load imbalances in the sequential computations affect the load balancing features of non-normal form program implementations, normal forms achieve a sensibly better service time.

Currently, we are investigating two different ways to extend the results discussed in this work. On the one hand, we are trying to evaluate the relationship between the number of resources (processing elements) needed to implement non-normal forms and the equivalent normal form programs. Experimental results show that the amount of resources needed to run normal form programs is close (usually smaller) to the amount of resources needed to run the equivalent non normal form programs achieving the best performance. On the other hand, we are currently investigating whether or not some kind of normal form can be found even in case that data *and* stream parallel skeletons are taken into account.

Acknowledgments We wish to thank Prof. Christian Lengauer for his useful comments to the first version of our work, and the Dept. of Computing, Imperial College, London, for the machines access.

References

[1] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations.* Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[2] J. Darlington, A. J. Field, P.G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming Using Skeleton Functions. In M. Reeve A. Bode and G. Wolf, editors, *PARLE'93 Parallel Architectures and Langauges Europe.* Springer Verlag, June 1993. LNCS No. 694.

[3] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220, July 1992.

[4] H. Burkhart and S. Gutzwiller. Steps Towards Reusability and Portability in Parallel Programming. In K. M. Decker and R. M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 147–157. Birkhauser, April 1994.

[5] P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Europar '96*, pages 601–614. Springer-Verlag, 1996.

[6] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. Parallel Computing, to appear, 1999.

[7] M. Vanneschi. PQE2000: HPC tools for industrial applications. *IEEE Concurrency*, (4):68–73, 1998.

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

[9] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1998.

[10] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based P3L compiler. In *Proceedings of the PCW'97*, 1997. Camberra, Australia.

[11] R. S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computing Science.* NATO ASI Series, 1988. International Summer School directed by F. L. Bauer, M. Broy, E. W. Dijkstra and C. A. R. Hoare.

[12] S. Gorlatch and C. Lengauer. (De)Composition Rules for Parallel Scan and Reduction. In *3rd Int. Conf. on Massively Parallel Programming Models*, November 1997. London.

[13] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: how to evaluate the data-parallel stream-paralle treadoff. In *Proceedings of the International Workshop on Constructive Methods for Parallel Programming*, 1998. Technical Report, University of Passau, No. MIP-9805.

[14] C. W. Kessler. Symbolic Array Data Flow Analysis and Pattern Recognition in Numerical Codes. In K. M. Decker and R. M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 57–68. Birkhauser, April 1994.

[15] B. Bacci, B. Cantalupo, P. Pesciullesi, R. Ravazzolo, A. Riaudo, and L. Vanneschi. SKIE: user's guide (version 2.0). Technical report, QSW Ltd. Rome, Italy, December 1998.

[16] C. T. King, W. H. Chou, and L. M. Ni. Pipelined data-parallel algorithms: Part I – Concept and Modeling. *IEEE Transactions on Parallel and Distributed Systems*, 1(4), October 1990.

[17] B. Bacci, M. Danelutto, S. Pelagatti, S. Orlando, and M. Vanneschi. Unbalanced Computations onto a Transputer Grid. In *Proceedings of The 1994 Transputer Research and Application Conference*, pages 268–282. IOS Press, October 1994. Athens, Georgia, USA.

[18] D. B. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising Data-Parallel Programs Using the BSP Cost Model. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing*, pages 698–703. Springer Verlag, 1998. LNCS No. 1470.

[19] C. B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *EuroPar'97 Parallel Processing*, pages 650–661. Springer Verlag, 1997. LNCS. No. 1300.