Alcom-FT Technical Report Series
ALCOMFT-TR-02-118

Grab and Go Systems: a CPO approach to concurrent web and grid-based computation *

J. Gabarró[1] and A. Stewart[2] and M. Clint[2]

[1] Dep. LSI, Universitat Politècnica de Catalunya, Jordi Girona, 1-3, Barcelona 08034, Spain.
[2] School of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland.

Abstract. A useful property of $web-based$ information systems [1] is the ability to display partial information. For example, a web program can display a sequence of fuzzy images which is extended by the production of improved images as execution of the program proceeds. A sequence of improving approximations to an image can be modelled by the elements of a complete partial order (CPO). CPOs can also be used to model *grid-based* computations. For example, a coalgorithm is a collection of iterative processes, each with identical functionality, which individually generate a series of improving approximations towards a desired goal. The aim of a coalgorithm is to produce faster convergence than could be achieved by any of its constituent processes separately. Each constituent process has the potential to exchange partial information with its companions so that all may make use of the best information available.
In this paper a *non-blocking* communication abstraction, based on CPOs, is used to develop a model of iterative web- and grid-based computations. The abstraction is novel in that it may not directly match a *send* communication in one process with a corresponding *receive* communication in another; rather, a *receive* communication is identified with taking the least upper bound of the set of messages available at an input port - this set may be empty, contain exactly one messsage or contain multiple messages. In all cases the receiver does not wait but gathers the best available information and proceeds with its computation. Thus, the abstraction corresponds to a loosely coupled model of distributed computation. The applicability of the model is illustrated by a number of disparate examples of distributed iterative computation.

Keywords web computation; grid computation; co-algorithms; CPOs; asynchronous non-blocking communications

1 Introduction

The notion of distributed computation has been modified as a result of recent developments in technology - thus, traditional models of distributed computation may not be the most appropriate to represent computations realised on web servers and computational grids. An essential difference between web- and grid-based computing and conventional distributed models is in the treatment of communication transfer rates. On a single machine communication time can be accurately predicted and it is normal for input communications either to receive a message if it is available or wait until the message arrives. In contrast, there may be wide variations in delivery time for web-based communications. In this paper an alternative non-blocking form of communication based on CPOs is proposed for iterative processes each of which generates a series of steadily improving information approximations. The overall computation may be accelerated by sharing, at certain points, the best available information over all of the processes.

Suppose that an iterative process S_{out} generates a series of improving approximations:

$$\perp \sqsubseteq f(\perp) \sqsubseteq f^2(\perp) \sqsubseteq \cdots \sqsubseteq f^k(\perp).$$

where $f^i(\perp)$ is the approximation generated by S_{out} on its ith iterative cycle, f is a monotonic function and $\perp$ is an initial approximation. After each approximation is generated it is sent asynchronously to another

* Work partially supported by: (1) The British Council and Spanish Ministry of Science and Technology ACI HB1999-0093, (2) Spanish CICYT under grant TIC-1999-0754-C03 (Mallba), (3) IST program of the EU under contract IST-2001-33116 (Flags), (4) Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

process S_{in}. For web- and grid-based computing it is desirable to decouple the system $S_{out} \parallel S_{in}$ as far as possible. To this end it is desirable to allow the generator process to be in a different iterative cycle from the receiver process. In a loosely coupled system the following situations may arise when the input port is inspected by S_{in}:

1. no information is available;
2. one piece of information is available; or
3. multiple pieces of information are available.

A new receive operation, *best_receive* is proposed which is defined for each of the cases above as follows:

1. S_{in} does not wait but accepts $\perp$;
2. S_{in} immediately accepts the only available datum; and
3. S_{in} immediately accepts the least upper bound ($\bigsqcup$) of the available approximations.

In this simple situation $\bigsqcup\{f^i(\perp)|0 \leq i \leq k\} = f^k(\perp)$. The situation may be extended to the case where there are multiple information generators - and so the least upper bound operation may have a more complex realisation. Alternatively, a generator may produce independent pieces of information which are not ordered. The novel feature of the communication is that the non-blocking *best_receive* instruction may be matched with none, one or multiple *send* instructions. Two concrete examples are given below.

Consider the problem of a client computer displaying and processing image information stored on a server. At a given instant the client may only have received partial information from the server. Nevertheless, it is desirable that this partial information be displayed and, perhaps, processed by a user. This simple situation can be modelled by a single chain of approximations as indicated above. Here $f^i(\perp) \sqsubseteq f^{i+1}(\perp)$ means that the ith approximation to the image, $f^i(\perp)$, is a subimage of the $(i+1)$th approximation, $f^{i+1}(\perp)$. When the client receives new input it can be compared against the available data and, if the new approximation is an improvement, the new image is displayed. This example can be easily extended to cover the situation where a video image is broadcast by a web server.

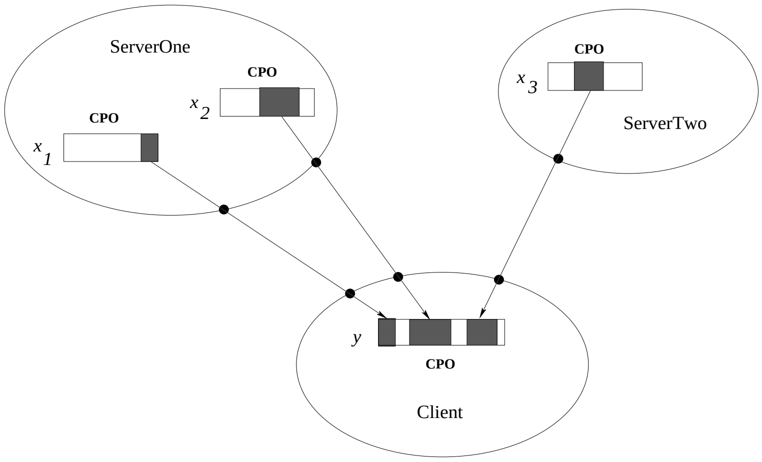

Fig. 1. *ServerOne* and *ServerTwo* send information to the *Client*. Shaded parts of the CPOs in the servers represent those parts of the CPO extracted by the client. The utilization of the information is realised by a function f to generate an approximation y viz $y = f(\Delta_1(x_1), \Delta_2(x_2), \Delta_3(x_3))$. Communication ports are represented by bullets. White areas in the client CPO represent local information.

The web example can be extended to the case where a *Client* process has to compute approximations from information selected from several servers (see Figure 1). Let x_j^i be the ith approximation in chain x_j, $1 \leq j \leq 3$ where a chain is a sequence of improving approximations. Suppose that the chains, x_j, are generated by remote servers and that a local client process wishes to process *parts* of the information supplied by these servers. Let Δ_j be an extraction function which can be applied to an element from chain x_j. Then the client may generate an approximation to:

$$f(\Delta_1(\lim_{i\to\infty} x_1^i), \Delta_2(\lim_{i\to\infty} x_2^i), \Delta_3(\lim_{i\to\infty} x_3^i))$$

from chains of approximations $x_j, 1 \leq j \leq 3$ provided that the extractor functions are *Scott continuous* [9] – that is

$$\Delta_j(\lim_{i\to\infty} x_j^i) = \lim_{i\to\infty}(\Delta_j(x_j^i)),\ 1 \leq j \leq 3$$

At any instant the server processes may have available only approximations, x_1^i, x_2^k, x_3^l, to the three CPO limits. f is required to be monotonic in all its arguments in order to ensure that an increase in the quality of the input approximations increases the quality of the client approximation.

It is widely recognised that the concept of *grid based computation* may result in radically different ways of organising computations. Inefficient algorithms for traditional architectures may be appropriate for implementation on computational grids. Consider a situation in which a problem may be solved by any one of three iterative algorithms. The convergence characteristics of the various algorithms may be data dependent, and so, it is not possible to say, in general, which algorithm has the best convergence behaviour. Further, one algorithm may converge uniformly while another may initially converge very slowly but, once a good approximate solution is found, may converge rapidly. It may be profitable for the algorithms to share information from time to time in order to improve overall performance. Suppose that the generated solutions are tuples of real numbers. Let S_1 generate approximations $\bigsqcup\{f^i(\bot,\ldots,\bot)|0 \leq i\}$, S_2 generate $\bigsqcup\{g^i(\bot,\ldots,\bot)|0 \leq i\}$ and S_3 generate $\bigsqcup\{h^i(\bot,\ldots,\bot)|0 \leq i\}$. At a given stage (or stages) in the process one algorithm may send an approximate solution to another algorithm. Let S_1 be on its ith iterative cycle, S_2 on its jth iterative cycle and S_3 on its kth iterative cycle when both S_1 and S_2 send information to S_3. Then, assuming no further information transfer, *best_receive* executed by S_3 accepts $\bigsqcup\{f^i(\bot,\ldots,\bot), g^j(\bot,\ldots,\bot)\}$. Here,

$$\bigsqcup\{(x_1,\ldots,x_n),(y_1,\ldots,y_n)\} = (\bigsqcup\{x_1,y_1\},\ldots,\bigsqcup\{x_n,y_n\})$$

A receiver may choose to ignore this approximation if it has a better one available locally; alternatively, the receiver may use the new partial information in its computation - perhaps by combining the best parts of its own data with the best parts of the other co-algorithm processes' data before proceeding with its computation. Thus, the best information available to S_3 is

$$\bigsqcup\{f^i(\bot,\ldots,\bot), g^j(\bot,\ldots,\bot), h^k(\bot,\ldots,\bot)\}$$

Both the web-based and grid-based examples require the processing of partial information. It is possible to model such computations by defining an information ordering and then requiring that computations are monotonic with respect to this order. In particular, the situations above can be modelled by representing communication ports by CPOs. This leads to the notion that a *receive* communication instruction does not wait for data but processes the least upper bound of the data available on the port - in the worst case a receive instruction returns the bottom element of the CPO (no data).

An abstract model of Grab and Go systems is developed in §2. The abstract model is illustrated by a number of concrete examples in §3 – these include web systems and coalgorithms suitable for grid implementation. A general discussion of the applicablity of approximation systems is given in §4.

2 Ports as CPOs

Asynchronous communications between members of a set of processes is represented by a port abstraction. A port is a data space shared among multiple producers and a single consumer. A port has an associated data structure – usually this is a FIFO queue so that the receiver can directly match inputs with outputs. Conventionally, a receive instruction applied to a non-empty port strips away the first-in datum and assigns it to a specified local variable. In contrast, Grab and Go communications are modelled by a port which retains only the best available information. Let $(D, \sqsubseteq, \bigsqcup)$ be a CPO [8] with domain D, ordering relation $\sqsubseteq$ and least upper bound operator $\bigsqcup$. Then $p : (D, \sqsubseteq, \bigsqcup)$ is a port of type D. The definitions of the port communication instructions *send* and *best_receive* are:

$$send(e,p) \quad =_{df} \quad p := \bigsqcup\{p\ ,\ e\ \} \tag{1}$$

$$best\_receive(x,p) \quad =_{df} \quad x := \bigsqcup\{p\ ,\ x\ \} \tag{2}$$

Here e is an expression and x is a variable name. Clearly the values of x and p are either improved or unchanged by port operations.

$$p \sqsubseteq send(e,p)_p$$

$$x \sqsubseteq best\_receive(x,p)_x$$

Here σ_z denotes the value of the variable z in the state σ. Further, sending $\perp$ (the least element of the CPO) has no effect:

$$send(\perp, p) = \Pi$$

where Π is the empty statement (SKIP). Similarly,

$$p = \perp \Rightarrow best\_receive(x,p) = \Pi$$

A system S utilising a port $p : (D, \sqsubseteq, \bigsqcup)$ may be defined as:

var $p : (D, \sqsubseteq, \bigsqcup) := \perp;$
S
end p

Here all communications through p are hidden from the environment – in this way Grab and Go systems are made non-reactive. It is assumed that $send(e,p)$ satisfies $e \in D$ and $best\_receive(x,p)$ satisfies $x \in D$. Further, $inport(S)$ and $outport(S)$ are used to denote the sets of ports used for input and output, respectively, by S. A port cannot be used for both input and output by the same process nor can it be used for input by more than one process:

$$inport(S) \cap outport(S) = \{\}$$

$$distinct(S_1, S_2) \Rightarrow inport(S_1) \cap inport(S_2) = \{\}$$

Note that a port may be initialised to a value in a CPO that is not its bottom element.

A model of the communication behaviour of a Grab and Go system can be devised by equating a multi-process system with a uni-processor implementation. The order in which the uni-processor executes instructions is not deterministic. For example, the system $send(e,p)\widetilde{\|}best\_receive(x,p)$ – where $\widetilde{\|}$ is the composition operator for systems with Grab and Go communications – may be modelled by executing the *send* instruction first; alternatively, the system can be modelled, with equal validity, by executing the *best_receive* instruction first. The interleaving model is complicated by the asynchronous nature of *send* communications. An instruction $send(e,p)$ may not *immediately* update port p - a delay of unknown duration may occur before the port is updated. Thus, unlike CSP, a *send* is not regarded as an atomic action without duration. A process cannot be identified with a single execution sequence; rather, a process may give rise to multiple execution sequences, each of which corresponds to a particular sequence of port update delays.

An abstract model of the interface between a port and its associated producer and consumer processes is now devised for $finite\ straight-line$ programs. A process S is deemed to have, for each of its output ports an associated *set* of communications, $S \uparrow p$. Let $p \in outport(S)$. Then:

$$S \uparrow p =_{def} \{a | \exists S_1, S_2 . S = S_1;\ send(a,p);\ S_2\}$$

Using a set to represent communications means that it is allowable for one communication to overtake another thereby modelling a network which delivers asynchronous communications in a manner that cannot be determined *a priori*. The set of communications associated with a given port p in system $S_1\widetilde{\|}S_2$ is:

$$(S_1\widetilde{\|}S_2) \uparrow p =_{def} (S_1 \uparrow p) \ \cup \ (S_2 \uparrow p)$$

Thus, for processes which only have output ports, $\widetilde{\|}$ is idempotent with respect to port updates:

$$(S\widetilde{\|}S) \uparrow p =_{def} (S \uparrow p)$$

Such a communication set *may* give rise to various sequences of port updates. Here, the trace notation of CSP [4, 5] is modified in order to define such sequences. In the context of approximation communications

a port trace is a sequence a values stored in the port. All such traces have a least element, say, d. Let $trace(p, d, A)$ denote the set of possible behaviours of port p where the port has a least element, d, and any incoming messages must be selected from the set A. Here t_0 and t' are, respectively, the first element of trace t and the tail of t. Then,
$trace(p, d, \{\}) =_{def} \{[d]\}$
$trace(p, d, A) =_{def} \{t | (t_0 = d \ \wedge \ \exists a \in A : t' \in trace(p, \bigsqcup\{d, a\}, A - \{a\})\}$
where $A \neq \{\}$.

Lemma 1. *$\forall t \in trace(p, d, A).chain(t)$.*

Proof. Let $\#t$ denote the length of trace t.
$\forall 1 \leq i \leq (\#t - 1) : t_i \sqsubseteq t_{i+1}$
$<$ by the definition of $trace$, $t_i \sqsubseteq \bigsqcup\{t_i, \ldots\} = t_{i+1} >$ □

Example 1. Consider the following straight-line process S:
$send(a, p)$; $send(b, p)$; $send(c, p)$. Then, $S \uparrow p = \{a, \ b, \ c\}$ and

$$\begin{aligned} trace(p, \perp, \{a, \ b, \ c\}) = \{ & [\perp, \ a, \ \bigsqcup\{a, b\}, \ \bigsqcup\{a, b, c\}], \\ & [\perp, \ a, \ \bigsqcup\{a, c\}, \ \bigsqcup\{a, b, c\}], \\ & [\perp, \ b, \ \bigsqcup\{b, a\}, \ \bigsqcup\{a, b, c\}], \\ & [\perp, \ b, \ \bigsqcup\{b, c\}, \ \bigsqcup\{a, b, c\}], \\ & [\perp, \ c, \ \bigsqcup\{c, a\}, \ \bigsqcup\{a, b, c\}], \\ & [\perp, \ c, \ \bigsqcup\{c, b\}, \ \bigsqcup\{a, b, c\}]\} \end{aligned}$$

Each of these traces is a chain. For example,

$$\perp \ \sqsubseteq \ a \ \sqsubseteq \ \bigsqcup\{a, b\} \ \sqsubseteq \ \bigsqcup\{a, b, c\}$$

□

A consumer in a system S may receive any trace t, $t \in trace(p, \perp, p \uparrow S)$ from port p. Such a trace is matched to the *best_receive* instructions, for port p, in the consumer process. Note, however, that there is no synchronisation in the interaction between the port trace and the input sequence. Thus, a trace, t, may give rise to the following set of input sequences of length i:

$$t \downarrow i =_{def} \{r \mid \exists j : 1 \leq j \leq \#t : r_0 = t_j \wedge r' \in t[j..\#t] \downarrow (i - 1)\}$$

Here $t[j..\#t]$ denotes the suffix of t whose first element is t_j. Note that $t \downarrow i$ allows elements of a port trace to be skipped over – this reflects the situation where the port is updated before its current data is read. Alternatively, the same element of a port trace may be read by successive *best_receive* instructions – this reflects the situation where a port is not updated between consecutive reads.

Lemma 2. *$\forall r \in t \downarrow i. \ chain(t) \Rightarrow chain(r)$.*

Proof. $\forall 1 \leq j < i : r_j = r_{j+1} \vee r_j = t_k \wedge r_{j+1} = t_l, \ l > k$.
The result follows for all chains t. □

Example 2. Consider the port trace $[\perp, \ a, \ \bigsqcup\{a, b\}]$. This trace may give rise to any of the following input sequences of length 2:
$[\perp, \ \perp]$, $[\perp, \ a]$, $[\perp, \ \bigsqcup\{a, b\}]$, $[a, \ \bigsqcup\{a, b\}]$, $[a, \ a]$, $[\bigsqcup\{a, b\}, \ \bigsqcup\{a, b\}]$. □

This highly non-deterministic behaviour is captured by the following *send* laws for straight-line processes:

$$\begin{aligned}
(send(e,p)\ ;\ send(e,p)) &= send(e,p) && (3)\\
send(e_1,p); send(e_2,p) &= send(e_2,p); send(e_1,p) && (4)\\
send(e_1,p); send(e_2,q) &= send(e_2,q); send(e_1,p) && (5)\\
x := f; send(e,p) &= send(e^x_f,p); x := f && (6)
\end{aligned}$$

In each case a *send* instruction gives rise to non-determinism – for example,

$$send(e_1,p); send(e_2,p) = send(e_1,p); send(e_2,p)\ \sqcap\ send(e_2,p); send(e_1,p)$$

Here $\sqcap$ denotes non-deterministic selection. Note that, in general:

$$(send(e_1,p)\ ;\ send(e_2,p))\ \neq\ send(\bigsqcup\{e_1,e_2\},p)$$

The communication sequence on the left hand side can be used to generate more execution sequences than can be constructed from the right hand side. The properties of the composition $send(e,p)\ ;\ best\_receive(x,q)$ can be derived by considering a *send* instruction to comprise separate evaluation and update actions:

$$send(e,p)\ =_{def}\ z := e; update(z,p)$$

Here z is a new variable name that cannot be used elsewhere and

$$update(z,p)\ =_{def}\ p := \bigsqcup\{p\ ,\ z\ \}$$

Then,

$$update(z,p)\ ;\ S\ =\ S\ ;\ update(z,p)$$

provided that z does not occur in S. Thus,

$$update(z,p)\ ;\ best\_receive(x,q)\ =\ best\_receive(x,q)\ ;\ update(z,p)$$

These rules ensure that the *dependent* composition $best\_receive(x,q)\ ;\ send(x,p)$ does not commute:

$$send(x,p)\ ;\ best\_receive(x,q) \sqsubseteq best\_receive(x,q)\ ;\ send(x,p)$$

A Grab and Go system is identified with all possible interleavings of the multiple execution sequences of its underlying processes. There is no requirement for an interleaving to synchronise input and output communications (unlike the situation in CSP where the composition operator imposes constraints, via shared actions, on the ways that the actions of processes are interleaved). The interleaved model enjoys the following properties:
$\widetilde{||}$ is commutative and associative:

$$\begin{aligned}
S_1 \widetilde{||} S_2 &= S_2 \widetilde{||} S_1 && (7)\\
S_1 \widetilde{||}(S_2 \widetilde{||} S_3) &= (S_1 \widetilde{||} S_2)\widetilde{||} S_3 && (8)
\end{aligned}$$

Grab and Go communications obey the following law:

$$(send(e,p)\widetilde{||}receive(x,p))\ =\ p,x := \bigsqcup\{p,e\}\ ,\ \sqcap\{\bigsqcup\{x,p,e\},\bigsqcup\{x,p\}\} \qquad (9)$$

Here the receiver may proceed if the message e is not available. Systems composed using the operator $\widetilde{||}$ are highly non-deterministic – in essence the non-determinism arises as a consequence of the very loose coupling between processes.

Example 3. Consider

$$S_1 = send(f(\bot), p) \ ; \ send(f^2(\bot), p)$$

and

$$S_2 = x := g(\bot) \ ; \ best\_receive(x, p).$$

All of the following outcomes are possible for the value of x in the final state of $S_1 \widetilde{||} S_2$:

- $x = g(\bot)$
- $x = \bigsqcup\{g(\bot), f(\bot)\}$
- $x = \bigsqcup\{g(\bot), f^2(\bot)\}$

□

Grab and Go systems cannot deadlock. Consider an empty process Π with access to a communication port p. Then,

$$\begin{aligned}(send(e,p)\widetilde{||}\Pi) &= send(e,p)\\ (best\_receive(x,p)\widetilde{||}\Pi) &= best\_receive(x,p)\end{aligned}$$

Again there are no requirements that an output communication be matched with an input communication.

Example 4.

$\quad$ **var** $p, q : (D, \sqsubseteq, \bigsqcup) := \bot,\bot;$

$\quad\quad send(f(\bot), p) \ ; \ best\_receive(x, q) \ \widetilde{||} \ send(g(\bot), q) \ ; \ best\_receive(y, p)$

$\quad$ **end** p

Here there is no requirement that the interleaved model should execute a *send* instruction first. Thus, the weakest interleaving arises when all *best_receive* instructions are executed before the execution of any *send* instructions. This is the worst-case situation in which all communications are delayed until after computation has terminated and so, for the weakest interleaving, $x = \bot \wedge y = \bot$ in the final state. □

One class of Grab and Go systems – those composed of *individually terminating* processes – is straightforward to reason about. Consider a system

$$(S_1(x_1)\widetilde{||}S_2(x_2)\widetilde{||}\cdots\widetilde{||}S_n(x_n))$$

where

1. the alphabet of each $S_i, 1 \le i \le n$ is $\alpha(S_i)$;
2. the input communication variable(s) of S_i is x_i;
3. each process $S_i, 1 \le i \le n$, is monotonic with respect to its input communication variables (i.e. $x \sqsubseteq y \Rightarrow (\forall z \in \alpha(S_i) : S_i(x)_z \sqsubseteq S_i(y)_z)$; and
4. each process $S_i, 1 \le i \le n$, is guaranteed to terminate irrespective of its input communications.

This system may give rise to non-deterministic behaviour. Let $||^I$ denotes independent composition. The weakest behaviour that the system may engage in is given by

$$(S_1' \ ||^I \ S_2' \ ||^I \cdots ||^I \ S_n')$$

where S' denotes the process S stripped of all *send* communications and with all input instructions of the form $best\_receive(x,p)$ replaced by the assignment $x := \bigsqcup\{x, \bot\}$. This new system corresponds to the worst case behaviour where all input communications arrive *after* all *best_receive* instructions have been executed. Thus, one way to reason about the original system is to consider only the worst case behaviour. The original system is guaranteed to produce the same or better information than in the worst case.

The case of potentially infinite processes is more complex.

Example 5. Consider

$$send(f(\bot), p)\widetilde{||}(x := \bot \ ; \ \textbf{while} \ (x \neq \bot) \ \textbf{do} \ best\_receive(x, p))$$

One possible interleaving of the processes S_1 and S_2 is generated from an infinite sequence of *best_receive* commands (from S_2) followed by the send instruction from S_1. This interleaving can be excluded by imposing a *fairness* constraint which necessitates that if a *send* instruction is offered then it will be executed after a finite amount of time. □

Further consideration of the validity of fairness constraints is deferred until the final section.

3 Concrete Examples

The abstract model of Grab and Go systems is made concrete through a number of examples choosen from web- and grid-based computing applications.

Example 6. Small_Web
Consider a system *Small_Web* which comprises a server process generating and sending a series of approximations to an image and a browser process which receives these approximations [3]. Given a set of image co-ordinates, $PIXEL$, and a set of colours, $COLOUR$, then an image has type $PIXEL\ X\ COLOUR$. As a result of the communications the browser may be able to generate images with higher degrees of resolution.

Small_Web$(k : int)$ $\equiv$
    var $p : (image, \subseteq, \cup) := \{\}$;
        $(Server(k)\ \widetilde{\|}\ Browser(k))$
    end p

The server program iterates k times and, on each iteration, sends a portion of the image to p. Let $recover(picture, k)$ select, according to the parameter k, a series of pixel/colour pairs from the image, *picture*:

Server(k: int) $\equiv$
    var i: int;
        for $i := 0$ **to** k **do**
            send($recover(picture, k)$, p)
        end
    end i

The *Browser* periodically inspects p and, if possible, improves its current approximation to the image:

Browser(k: int) $\equiv$
    var y: image, i: int:=$\{\}, 0$;
        for $i := 0$ **to** k **do**
            $best\_receive(y, p)$;
            display(y)
        end
    end y, i

Here the least upper bound operation is set union. Note that the series of approximations generated by the server may be disjoint and that the sequence of approximations stored by the port is a chain. □

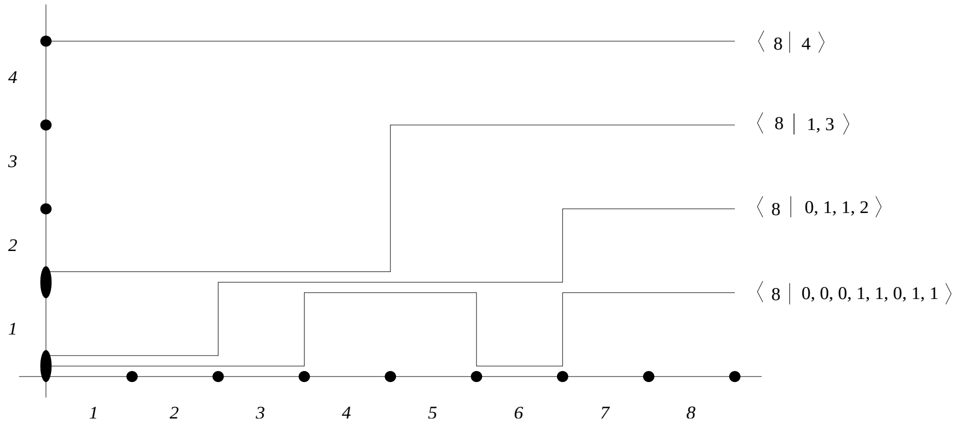

Fig. 2. Scaled version of bipmap approximations to the image $[0, 0, 0, 1, 1, 0, 1, 1]$. The scaling has been carried out in order to simplify the presentation. The picture corresponds to the sequence: $\langle 8 \mid 4\rangle \sqsubseteq \langle 8 \mid 1, 3\rangle \sqsubseteq \langle 8 \mid 0, 1, 1, 2\rangle \sqsubseteq \langle 8 \mid 0, 0, 0, 1, 1, 0, 1, 1\rangle$. The display corresponding to: $\langle 8 \mid 4\rangle$ is scaled by a factor 8, $\langle 8 \mid 1, 3\rangle$ is scaled by a factor 4 and $\langle 8 \mid 0, 1, 1, 2\rangle$ is scaled by a factor 2.

Example 7. Bitmap Images
In a variant of the preceding example an image may be represented by bitmap approximations. Each approximation consists of the following information:

$$\langle \text{total\_size} \mid \text{grey\_level}, \text{grey\_level} \ldots \rangle$$

For instance, the image $[0, 0, 0, 1, 1, 0, 1, 1]$ may be represented by the following sequence of approximations (see Fig 2):

$$\langle 8 \mid 4 \rangle \sqsubseteq \langle 8 \mid 1, 3 \rangle \sqsubseteq \langle 8 \mid 0, 1, 1, 2 \rangle \sqsubseteq \langle 8 \mid 0, 0, 0, 1, 1, 0, 1, 1 \rangle.$$

The first approximation could be displayed by averaging the value 4 over 8 points while the second approximation could be displayed by approximating the first four points by the value 0.25 and the second 4 points by the value .75. Thus, the degree of resolution of the picture increases as the accuracy of the approximations improves. The ordering relation, $\sqsubseteq$, is reflexive and transitive and satisfies:

$$\langle n \mid u + v, w + x \rangle \sqsubseteq \langle n \mid u, v, \ldots, w, x \rangle$$

Note that $bitmap_1 \sqsubseteq bitmap_2 \Rightarrow \bigsqcup\{bitmap_1, bitmap_2\} = bitmap_2$ □

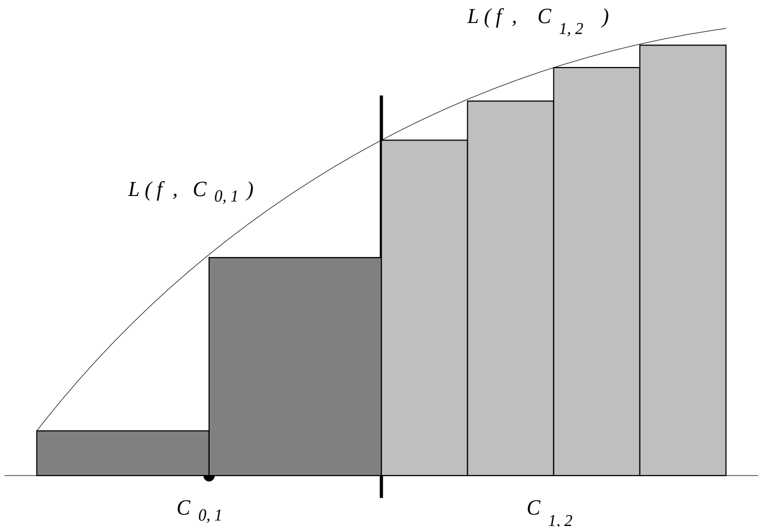

Fig. 3. An illustration of an approximation to $\int_0^2 f(x)dx$. Here $\int_0^1 f(x)dx$ is approximated by $\mathcal{L}(f, \mathcal{C}_{0,1}) = 1/2(f(0) + f(1/2))$ and $\int_1^2 f(x)dx$ is approximated by $\mathcal{L}(f, \mathcal{C}_{1,2}) = 1/4(f(1) + f(5/4) + f(3/4) + f(7/4))$.

Example 8. Display_Integral
Consider a system *Display_Integral* which computes and displays approximations to the integral [10]

$$\int_0^2 f(x)dx,$$

using the property

$$\int_0^2 f(x)dx = \int_0^1 f(x)dx + \int_1^2 f(x)dx.$$

Before considering the construction of the system some preliminary definitions of intervals and integral approximations are given. $x \in \{0, 1\}^\ell$ denotes that x is a binary string of length ℓ. Let $x = x_1 x_2 \ldots x_\ell$. Then, given a non-negative integer, n, $n.x = n.x_1 x_2 \ldots x_\ell$ denotes the number

$$n + x_1 2^{-1} + x_2 2^{-2} + \cdots + x_\ell 2^{-\ell}.$$

The interval $[n \mathrel{..} n+1]$ may be characterised by a family of partitions, $\{\mathcal{C}_{n,\ell} | \ell \geq 0\}$ where

$$\mathcal{C}_{n,\ell} = \{[n.x \mathrel{..} n.x + 2^{-\ell}] \mid x \in \{0,1\}^{\ell}\},$$

For example,

$$\begin{aligned}\mathcal{C}_{0,2} &= \{[0.x \mathrel{..} 0.x + 1/4] \mid x \in \{0,1\}^2\} \\ &= \{[0.00 \mathrel{..} 0.00 + 1/4], [0.01 \mathrel{..} 0.01 + 1/4], [0.10 \mathrel{..} 0.10 + 1/4], \ldots\} \\ &= \{[0 \mathrel{..} 1/4], [1/4 \mathrel{..} 1/2], [1/2 \mathrel{..} 3/4], \ldots\}\end{aligned}$$

The sequence $\{\ell \mapsto \mathcal{C}_{n,\ell} | \ell \geq 0\}$ expands to:

$$\begin{aligned}\mathcal{C}_{0,0} &= \{[0 \mathrel{..} 1]\}, \\ \mathcal{C}_{0,1} &= \{[0 \mathrel{..} 1/2], [1/2 \mathrel{..} 1]\}, \\ \mathcal{C}_{0,2} &= \{[0 \mathrel{..} 1/4], [1/4 \mathrel{..} 1/2], [1/2 \mathrel{..} 3/4], [3/4 \mathrel{..} 1]\}, \\ &\vdots\end{aligned}$$

In going from $\mathcal{C}_{0,1}$ to $\mathcal{C}_{0,2}$ the subinterval $[0 \mathrel{..} 1/2]$ is refined into the pair $[0 \mathrel{..} 1/4]$, $[1/4 \mathrel{..} 1/2]$. In Fig 3 there is a partition of $[0 \mathrel{..} 1]$ by $\mathcal{C}_{0,1}$ and a partition of $[1 \mathrel{..} 2]$ by $\mathcal{C}_{1,2}$. Given a continous non-decreasing and non-negative function f over the interval $[n \mathrel{..} n+1]$ and a partition $\mathcal{C}_{n,\ell}$ a lower bound on the integral above is given by the Riemann lower sum (see Fig 3 for examples of Riemann sums):

$$\mathcal{L}(f, \mathcal{C}_{n,\ell}) = \sum_{x \in \{0,1\}^{\ell}} f(n.x) 2^{-\ell}.$$

It follows that

$$\mathcal{L}(f, \mathcal{C}_{n,\ell}) \leq \int_n^{n+1} f(x)dx,$$

If $\ell \leq \ell'$ then it follows that $\mathcal{L}(f, \mathcal{C}_{n,\ell}) \leq \mathcal{L}(f, \mathcal{C}_{n,\ell'})$. A partial order for integral approximations is $(\Re^*, \leq, max)$ where $\Re^*$ denotes the non-negative reals; the bottom element of the order is 0.

The system *Display_Integral* consists of three processes:

Display_Integral(k) $\equiv$
 var p, q : $(\Re^*, \leq, max) := 0.0, 0.0$;
  (*Riemann_Sum*$(0, k, p)$ $\widetilde{\|}$ *Riemann_Sum*$(1, k, q)$ $\widetilde{\|}$ *Collect_Sum*(k, p, q))
 end p, q

Riemann_Sum(n, k, p) sends the first k Riemann approximations to $\int_n^{n+1} f(x)dx$ to port p:

Riemann_Sum$(n, k$: int, $p : (\Re^*, \leq, max))$ $\equiv$
 var i: int;
  for $i := 0$ **to** k **do**
   send($\mathcal{L}(f, \mathcal{C}_{n,i}), p$);
  end
 end i

Riemann_Sum sends the chain of messages:

$$\mathcal{L}(f, \mathcal{C}_{n,0}) \sqsubseteq \mathcal{L}(f, \mathcal{C}_{n,1}) \sqsubseteq \cdots \sqsubseteq \mathcal{L}(f, \mathcal{C}_{n,k}).$$

From a mathematical point of view, it is sufficient to communicate only the last value $\mathcal{L}(f, \mathcal{C}_{n,k})$. However, from a computational point of view, the earlier terms in the series have less complexity and can be generated more quickly. From a web perspective it may be advantageous to send approximate information to a client rather than waiting until an accurate solution is available.

Process *Collect_Sum* collects and displays two series of Riemann sums approximations from ports p and q, respectively (see Fig 3 for and example of a displayed value):

```
Collect_Sum(k: int) ≡
    var firstHalf, secondHalf: ℜ*, i: int := 0.0,0.0,0;
        for i := 0 to k do
            best_receive(firstHalf, p);
            best_receive(secondHalf, q);
            display(firstHalf + secondHalf)
        end;
    end firstHalf, secondHalf,i
```

Lemma 3. *System Display_Integral displays a chain of values taken from:*

$$\{\mathcal{L}(f, \mathcal{C}_{0,i}) + \mathcal{L}(f, \mathcal{C}_{1,j}) \mid -1 \leq i, j \leq k\}.$$

where $\mathcal{C}_{n,-1} = 0.0$

Proof. *Riemann_Sum*$(0, k, p)$ and *Riemann_Sum*$(1, k, q)$ generate the following sequences of (*send*) communications on ports p and q respectively:

$$\{\mathcal{L}(f, \mathcal{C}_{0,i}) \mid -1 \leq i \leq k\}$$

$$\{\mathcal{L}(f, \mathcal{C}_{1,j}) \mid -1 \leq j \leq k\}$$

These sequences may be permuted and interleaved to model the actual order of arrival of messages in the collector processor. Let $\mathcal{C}_{n,-1}$ denote the bottom element of the ports p and q (0.0). Then the system behaviour is determined by all interleavings of the $2k$ *best_receive* instructions with the permuted and interleaved input sequences. For example, let v^i denote the value of variable v in iteration i of the collector process. Then

$$firstHalf^0 = \sqcap\{\mathcal{L}(f, \mathcal{C}_{0,i}) \mid -1 \leq i \leq k\}$$

and

$$secondHalf^0 = \sqcap\{\mathcal{L}(f, \mathcal{C}_{1,j}) \mid -1 \leq j \leq k\}$$

Further,

$$firstHalf^0 + secondHalf^0 \sqsubseteq \cdots \sqsubseteq firstHalf^k + secondHalf^k$$

□

In the worst case the *best_receive* instructions are executed before the *send* instructions and so only the weakest approximation, viz 0.0, is displayed. In order to avoid this possibility, termination of the collector process could be made dependent on the value of the current integral approximation. For example, the generators could be extended to send a series of both lower and upper bounds. The collector could have a termination condition which requires that the combined lower and combined upper approximations differ by, at most, a prescribed tolerance. □

Example 9. Eigensystem Coalgorithm
Consider a co-algorithm to determine the m $(\leq n)$ dominant eigenpairs of $A \in \Re^{n \times n}$. An eigenvalue-eigenvector pair (λ, x) of A satisfies:

$$Ax = \lambda x$$

Computing subsets of eigenvalues and eigenvectors of very large, often sparse, matrices is an important feature of many important applications. The only feasible way to perform this task is to use an iterative method in which the matrix of interest, A, is used solely as a multiplier. A number of independent iterative algorithms are available for the solution of the problem. The coalgorithm proposed here is constructed from two of these: Lanczos' method and subspace iteration [2]. For both algorithms it is assumed that the eigenpairs are extracted in order of decreasing eigenvalue magnitude. This assumption simplifies the treatment of communication ports below. The variant of Lanczos' method that is considered here computes the eigenvectors one at a time [11]; when an eigenvector has been isolated the method is restarted with a new vector (which is orthogonal to previously computed eigenvectors). A sequential version of the algorithm is:

$Lanczos(A\text{: } \Re^{nX\ n},\ X\text{: Set of } (\Re^n),\ x : \Re^n\ ) \ \equiv$
$X \lhd \#X = m \rhd$
$\quad (Lanczos(A, X \cup \{x\}, \text{ortho}(e_{\#X+2},\ X \cup \{x\}))\lhd \text{ converged}(x) \ \rhd$
$\qquad Lanczos(A, X, f(x)))$

Here ortho$(y,\ X)$ orthogonalises a vector y with respect to the set of vectors X, e_i deotes the unit vector of length n whose ith component is 1, converged(x) holds if x is a sufficiently accurate approximation to an eigenvector of A and f denotes the iterative body of Lanczos' method.

A second method –subspace iteration – repeatedly refines m approximations to the required eigenvectors. Each iteration of this method is more computationally intensive than an iteration of Lanczos' method but its storage requirements are fixed. On each iteration approximations to all required m eigenpairs are available.

$Subspace(A\text{: } \Re^{nX\ n},\ x_1 \ldots x_m : \Re^n\ ) \ \equiv$
$\quad x_1 \ldots x_m \lhd \text{ converged}(x_1 \ldots x_m) \ \rhd Subspace(A, g(x_1 \ldots x_m))$

Here g denotes the iterative body of subspace iteration.

There are many ways in which to construct coalgorithms based on the methods above. Coalgorithms use partial eigenpair information to improve the overall convergence rates. Here, only a simple coalgorithm is presented which uses information generated by subspace iteration to improve the convergence rate of Lanczos' method. A CPO for ordering vectors can be constructed using a metric $\mathbf{M}$:

$$\mathbf{M}(x) =_{def} \| \ (Ax - \overline{\lambda}x)\ \|_2$$

Here the eigenvalue approximation $\overline{\lambda}$ is given by:

$$\overline{\lambda} = x^T A x$$

and the norm $\|\ x\ \|_2$ is defined as $\sqrt{x^T x}$. The vector approximations x and y are ordered according to $x \sqsubseteq y$ iff $\mathbf{M}(x) \geq \mathbf{M}(y)$. The least upper bound of two vectors x and y is given by:

$$\bigsqcup\{x, y\} =_{def} \begin{cases} x \text{ if } \mathbf{M}(x) \leq \mathbf{M}(y) \\ y \text{ otherwise} \end{cases}$$

Then a co-operating system which computes the m dominant eigenvectors of A is:

$CoEigenAlgorithms(A\text{: } \Re^{n\ X\ n}) \equiv$
$\quad$ **var** $Lp_1,\ \ldots,\ Lp_m : (\Re^n, \sqsubseteq, \bigsqcup) := e_1,\ \ldots,\ e_m$
$\qquad CoLanczos(A,\ \{\}, e_1)\ \widetilde{\|}\ CoSubspace(A,\ e_1, \ldots, e_m)$
$\quad$ **end** $Lp_1, \ldots, Lp_m$

In the coalgorithm version of subspace iteration, $CoSubspace$, the best available information about the eigenvectors of A is sent to the ports $Lp_1,\ \ldots,\ Lp_m$:

$CoSubspace(A\text{: } \Re^{n\ X\ n},\ x_1 \ldots x_m : \Re^n\ ) \ \equiv$
$\quad x_1 \ldots x_m \lhd \text{ converged}(x_1 \ldots x_m) \ \rhd$
$\qquad$ **for** $i := 1$ **to** $m\ send(x_i, Lp_i);\ CoSubspace(A, g(x_1 \ldots x_m))$

$CoLanczos$ may use port information to improve its convergence rate:

$CoLanczos(A\text{: } \Re^{n\ X\ n},\ X\text{: Set of } (\Re^n),\ x : \Re^n\ ) \ \equiv$
$X \lhd \#X = m \rhd$
$\quad (X := X \cup \{x\};\ x := e_{\#X+1};$
$\quad best\_receive(x,\ Lp_{\#X+2});$
$\quad CoLanczos(A, X, \text{ortho}(y,\ X)))\lhd \text{ converged}(x) \ \rhd$
$\qquad best\_receive(x,\ Lp_{\#X+1});\ CoLanczos(A, X, f(x)))$

Here $CoSubSpace$ is used, where possible, to improve $CoLanczos$' current eigenvector approximation, or, in the case of a restart, to provide an initial starting vector. □

4 Discussion

Web-based computing is characterised by sets of participating processes sharing information over unpredictable communication networks. In the worst case such systems must function with broken links. Information on server pages may be delivered in discrete stages and so a client process may, at times, have only partial information available. Similarly, a client process may not wait for complete information before passing on its data to a user. In order to capture the essence of web-based computing a novel loosely coupled model of distributed computing based on CPOs has been presented. The model does not impose fairness constraints – this reflects the possibility of broken links in a network. It transpires that the model which was developed to describe web-computation is also appropriate for the description of certain kinds of grid-based computations.

The advent of Grid computing will lead to a re-evaluation of how best to build applications software which can effectively exploit heterogeneous, widely dispersed resources. In many important current applications, in which heavy use is made of numerical linear algebra, the volume of data to be processed and the sparsity of the very large matrices involved points to the use of iterative methods as the only feasible approach to the solution of certain constituent problems – such as the solution of linear equations and the computation of (partial) eigensystems. There is a need to reappraise these methods in the light of the sorts of facilities which the Grid provides. In particular, the use of polyalgorithms [7] and coalgorithms offers the possibility of improved performance in a Grid environment. In a coalgorithm, a collection of iterative algorithms, each having the same functionality, periodically assist each other in the solution of a problem by supplying to one another their current approximate solutions in the hope that (at least) one of them (not necessarily the sender) can exploit the best of these better than any of the others. The coalgorithm approach is elegantly modelled by the proposed port-based computational model. The approach has been illustrated by a (simple) coalgorithm for the computation of a partial eigensolution of a real symmetric matrix.

A model of only $finite\ straight - line$ processes has been presented here. Thus, a number of questions remain open, including:

1. **treatment of infinite processes**: is a $fairness$ constraint needed to ensure that an infinite sequence of *send* instructions cannot, in its entirety, overtake an infinite sequence of *best_receive* instructions in the interleaved model?
2. **implementation** can an efficient implementation of the model be devised? At present MPI [6] supports asynchronous non-blocking send and receive instructions. Can efficient implementations of *send* and *best_receive* be developed from their non-blocking MPI counterparts?

References

1. Baeza-Yates R., Ribeiro-Neto B.: Modern Information Retrieval, Addison Wesley, 1999.
2. Demmel J. W.: Applied Numerical Linear Algebra, SIAM, 1997.
3. Gibbs S.J., Tsichitzis D.C.: Multimedia Programming Addison Wesley, 1994.
4. Hoare C. A. R.: Communicating Sequential Processes, Prentice Hall, 1985.
5. Hoare C. A. R., Jefeng H.: Unifying theories of programming, Prentice Hall, 1998.
6. MPI: A Message-Passing Interface Standard. University of Tennessee, 1995.
7. Rice J.R.: On the construction of polyalgorithms for automatic numerical analysis. In Interactive Systems for Experimental Applied Mathematics (M. Klerer and J. Reinfelds, eds.) Academic Press , 301–313, 1968.
8. Schmidt D.A.: Denotational Semantics, Allyn and Bacon, 1986.
9. Stoy J.E.: Denotational Semantics: The Scott-Strachey approach to programming language theory, MIT press, 1977.
10. Daniel W. Stroock, D.W.: A concise introduction to the theory of integration, World Scientific 1990.
11. Szularz M., Weston J., Clint M.: Explicitly restarted Lanczos algorithms in an MPP environment, Parallel Computing, Vol 25, No. 5, 613–631, 1999.