UNIVERSITE D'ORLEANS
Faculté des Sciences
LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

A Cost Model For Asynchronous and Structured Message Passing (Extended Version) [a]

Emmanuel Melin, Bruno Raffin,
Xavier Rebeuf and Bernard Virot
Université d'Orléans, LIFO

Rapport N^o **99-07**

[a]This work has been partly supported by the "Institut du Développement et des Ressources en Informatique Scientifique" (IDRIS), Orsay, France. Project number 970829.

Abstract

In this paper, we present a cost model taking into account properties of present time machines. This work relies on $\mathcal{SCL}$-chan, an asynchronous parallel execution model allowing explicit message passing. We show that it is possible to define a complexity function for $\mathcal{SCL}$-chan programs. It yields a symbolic date for each communication event. These dates can be ordered to compute upper bounds for the network load. In contrast to other classical approaches this cost computation can handle asynchronism in message passing and communication/computation overlap.
Keywords: Cost Model; Symbolic Date; Parallel Programming Languages; Asynchronous Execution; Structural Clock.

1 Introduction

In parallel programming, the main difficulty is to find an algorithm solving a given problem with the "best" possible speed up. To compare two algorithms, a natural method consists in comparing their execution times. Nevertheless, execution time is not a good criterion since it depends on the characteristics of the target machine. Therefore, this is an important problem to define a model of parallel complexity to quantify a priori the efficiency of an algorithm.

Several complexity models have been proposed for synchronous computation models, but the prevailing model is the PRAM one [10]. All processors are synchronized after each elementary computation step and they communicate through a shared memory. The cost associated to an algorithm is the number of synchronous elementary steps performed. The PRAM model allows straightforward cost evaluation for complex algorithms. Nevertheless, it is not well suited for actual distributed memory machines since it does not take into account communication overhead.

Valiant and Mc Coll [17, 11] propose another approach. They introduce a cost model called BSP. A computation is a sequence of supersteps. Inside of a superstep, each processor performs local computations and sends messages. Each superstep is completed by a global synchronization to validate the communications for the next superstep. The cost associated to each superstep takes into account local computations, communications and the global synchronization. Thanks to the global synchronization, the communication overhead is globally considered for each superstep. The communication overhead for a given superstep only depends on the computer characteristics and on the number of messages sent in the superstep. Therefore, the BSP model yields a simple programming and cost model. The prediction of performance is quite accurate in many cases. Nevertheless, the mandatory use of global synchronizations can lead to inefficient programs.

Skillicorn [16] proposes a more asynchronous version of BSP called MINIBSP. The MINIBSP composition rules allow supersteps to be composed sequentially and in parallel. Therefore, it is a subset-synchronizing language. Nevertheless, partial synchronizations may introduce useless waiting points for some processors.

Culler and Al [2] define a model called LogP, well suited for asynchronous algorithms. The model is based on four parameters which abstractly specify abstractly the target machine. It analyzes the communication scheme in order to deduce the cost associated. Nevertheless, the communication scheme only captures a specific execution of the program, thereby making difficult the evaluation of "worst case" costs.

Our goal is to show that it is possible to develop a complexity model leading to an a priori cost evaluation well suited for asynchronous message passing. An asynchronous execution of a parallel program can be viewed as a DAG; where a node represents the execution of an instance of instruction by a single processor and an arrow symbolizes a sequence. The order defined by this DAG has to respect causality. It can be total (as in the PRAM model) or partial (as in the BSP and LogP models). The weights on arrows depend on the modelization of the architecture. The cost associated to the execution is then the longest path in the graph.

To compute a worst case cost, one can think of two different methods. The first method consists in considering the algorithm DAG as the gathering of all possible DAGs. Nevertheless, gathering numerous DAGs can lead to a combinatorial explosion. To limit the number of node of the DAG, we only consider one node by instance of instructions. We construct then a graph that subsumes all the execution DAGs by adding sequence arrows corresponding to more possible dependencies. With classical message passing models, appending arrows can result in cycles, thereby forbidding the computation of a longest path.

In previous papers [15], we have defined a communication model called $\mathcal{SCL}$-chan. Communications take place through channels and rely on explicit send/receive instructions. The communication semantics relies on the syntactic structure of the program. A receive instruction can only refers to a send taking place before in the program structure. In contrast to classical message passing models, synchronizations and communications are dissociated. A receive instruction does not block on empty channel, but it waits for the sending processor to perform all instructions appearing before the receive statement in the program structure. As send/receive matching is not mandatory, it becomes possible to express easily irregular communication schemes.

In this paper, we show that it is possible to define a cost model for $\mathcal{SCL}$-chan programs. We propose a complexity function based on an induction on the program structure. It yields a symbolic date for each communication event. These dates can be ordered, allowing to compute upper bounds for the network load corresponding to each of the dates. In contrast to other classical approaches such as BSP, the cost computation can handle asynchronism in message passing and communication/computation overlap. Moreover, The structured communication model ensures that the algorithms DAGs subsuming all the execution DAG cannot contain cycle. Therefore, the complexity model can always compute the cost associated to the algorithm.

We first present the $\mathcal{SCL}$-chan language. Next, we develop our complexity model. Finally, we illustrate the capabilities of this model through several examples.

2 A structured message passing language

2.1 The $\mathcal{SCL}$-chan abstract machine

To preserve the compositionality, we have to give up unstructured asynchronous message passing models like MPI. In [15], we defined a model called $\mathcal{SCL}$-chan based on a communication semantics respecting the program structure. For sake of completeness, we briefly recall this model.

The program structure is an information shared by all indices. We impose that a receive instruction can only refers to a send taking place before in the program structure.

The abstract machine is an asynchronous parallel one with P processors. Processors are also called *indices* and denoted by u, v,... Each index u owns a *(private) memory* where it stores its own data. Memory allocation is uniform, i.e. each variable X is systematically allocated in each private memory. The value of a variable X stored in the private memory of an index u is denoted by $X|_u$. These data cannot be directly referenced by another index.

Data exchanges between indices take place through explicit send/receive communications. Therefore, a data of the private memory of an index u becomes available for a distant index v only if this data is explicitly *sent* to v. The index v can record such a value in its private memory only if it explicitly *receives* it. The data in transit, i.e. the data sent and not yet received are stored in *channels*.

There exists exactly one channel by variable and by pair of indices. Each channel corresponds to a LIFO (Last-In First-Out) list managed by a specific ordering we present in the next section. We denote by (u, v, X) the channel associated to the sender u, the receiver v and the variable X. At the beginning of a $\mathcal{SCL}$-chan program all channels are empty.

2.2 Language instructions

The asynchronous execution model of $\mathcal{SCL}$-chan relies on SPMD paradigm. All the indices execute the same program. An index that executes an instance of an instruction is called *active* for this instance. Indices execute instructions asynchronously. Only the receive instruction can block.

No action: skip. This instruction does nothing and just terminates.

Assignment: $X := E$. An active index u updates its value $X|_u$ of X in its private memory with the value $E|_u$. The expression E must be *pure*, i.e. evaluable in the private memory of the index u.

Synchronous sequence: $S; T$. When an active index finishes the execution of S, it starts the execution of T.

Concurrent conditioning: where B do S elsewhere T end. If an active index u evaluates the pure condition $B|_u$ to *true*, then it executes the instruction block S, otherwise it executes T.

Remark that a single branch conditioning construct where B do S end may be defined as where B do S elsewhere skip end.

Loop: loopwhere B do S end. Iteration is expressed by classical loop unfolding. An active index repeatedly executes S while it evaluates the pure expression $B|_u$ to *true*.

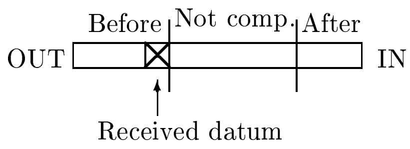
Send: send X to A. This instruction is non blocking. The expression A must be pure. An active index u sends the value $X|_u$ to the index $v = A|_u$, i.e. it adds this value to the channel (u, v, X).

Receive: receive X from A into Y. The expression A must be a pure. The semantics of this instruction relies on a logical partial ordering on the instances of program instructions. Hence, we first recall this ordering definition.

Definition 1 (Instruction ordering) *An instance of an instruction i_1 is* **before** *an instance of an instruction i_2 if there exists at least an active index for i_1 that can become later active for i_2. If there exists no index executing both i_1 and i_2, then these instruction instances are* **not comparable**.

Examples.

- Consider the program $X := 1;$ loopwhere $X < 10$ do $X := X + 1;$ end. The instruction $X := 1$ comes before the first instance of the instruction $X := X + 1$ (the instruction executed at the first loop iteration) which in turn comes before the second instance of $X := X + 1$.
- Two instructions, each one belonging to a different branch of the same where/elsewhere construct, are not comparable.



We turn now to the informal semantics of the receive X from A into Y instruction. The receive is a blocking instruction. Its execution depends on a *waiting condition*: an active index u waits that the sending index $v = A|_u$ has executed all the send instructions coming before this receive (in the sense of the ordering previously defined). When this condition is satisfied, the data of the channel (u, v, X) can be divided into three sets BE, NC and AF. The set BE is composed of the data sent by instances of send instructions coming before the receive, the set NC corresponds to the data sent by send instructions not comparable and the set AF contains the data sent by send coming after (see the above figure). If the set BE is empty, the index u assigns `nil` to $Y|_u$. Otherwise, it first selects the most recent datum of BE, next it assigns it to $Y|_u$ and last it removes all the data of BE. Therefore, channels are managed as LIFO lists with respect to the instruction ordering.

Remarks:

- Because two instructions, each one belonging to a different branch of the same where/elsewhere construct are not comparable, no communication can take place between these two branches. The where/elsewhere expresses disjoint parallelism [1]. A datum sent from one branch to the other can only be received by a receive executed after exiting the where/elsewhere. It can be seen as if these send have been performed just at the end of the where/elsewhere construct.
- Communication semantics ensures that a receive instruction cannot refer to a send not comparable or coming after it with regard to the logical ordering on instructions.
- Exact matching between send and receive instructions is not mandatory. More send than receive can be executed on a same channel or vice versa. This feature is important as it allows the programmer to limit communications when they are not foreseeable [13]. Classical message passing languages and libraries [6, 7] do not offer such a possibility.

2.3 Structural Clocks

To be feasible, the asynchronous execution model requires that each index should know its position in the program structure according to the instance of the instruction currently executed. Therefore, an index that executes a send instruction can stamp the send data by its current position. An index executing a receive instruction knows if it needs to wait for the sending index. Next, it can use the stamps of data in channels to complete the receive. To encode index positions, we introduce the *Structural Clocks* [9, 8, 12, 13]. Structural clocks record index positions with regard to the instances of instructions that store data in the channels, that is send instructions. Because the execution of these instructions can depend on enclosing control structures, the instructions taken into account, called the *meaningful* ones, are send, loopwhere and where/elsewhere. The Structural clock t_u of an index u is expressed by a list of pairs. Each term of the list corresponds to a nesting level. Each pair (l, c) is composed of a label l and an instruction counter c. The counter c represents the number of meaningful instructions already executed in the corresponding instruction block. The label l is used to distinguish which branch of a where/elsewhere statement an index is inside. Lists are built up by popping or pushing pairs on their right hand side. Note that we do not have to consider each iteration of a loopwhere instruction as a new nesting level. We only push a new term on the list at the first loop iteration and next, we count the instructions already executed in the loopwhere body.

With the aim of ordering two structural clocks, we compare the local counters corresponding to the common instruction block of the innermost nested level. We so define this partial ordering as a lexicographical ordering based on partially ordered pairs.

Definition 2 *Let u and v be two indices with their structural clocks t_u and t_v. An index u is said to be* **before** *an other index v (denoted by $t_u \prec t_v$) if one of the following conditions holds:*

- *there exists t not empty such as $t_v = t_u\ t$;*
- *there exists $t^1,\ t^2,\ t^3$ (possibly empty) and $c_u,\ c_v,\ l$ such as $t_u = t^1(l, c_u)t^2,\ t_v = t^1(l, c_v)t^3$ and $c_u < c_v$.*

If we consider only meaningful instructions, this ordering formalizes the one defined in Definition 1. The index u is *before* v ($t_u \prec t_v$) if and only if the instruction instance executed by u is *before* the one executed by v.

Example. The following example illustrates structural clock management. We display each instruction and the resulting clock after its execution.

Program	*Clocks*
$X := 1;$	$(0,0)$
send X to $This + 1;$	$(0,1)$
where $This\ mod\ 2\ =\ 0$ do	$(0,1)(1,0)$
$X := X + 1;$	$(0,1)(1,0)$
send X to $This + 1$	$(0,1)(1,1)$
elsewhere	$(0,1)(2,0)$
receive X from $This - 1$ into X	$(0,1)(2,1)$
end	$(0,2)$

The receive instruction cannot match the second send because they belong to different branches of the where structure. Thus, their clocks are not comparable. The receive can only match the first send instruction. Therefore, the received value is 1.

3 A complexity function for $\mathcal{SCL}$-chan programs

We turn now to the definition of a complexity function for $\mathcal{SCL}$-chan program.

3.1 The transmission time

To take into account the communication overhead, we have to compute the transmission time of a datum. As BSP, we model the network by a router. For sake of simplicity, we assume that the transmission time of a message does not depend on indices positions. Therefore, this duration only depends on a start-up phase and on the current network load.

We define our time unit to be the time required the execution of an elementary instruction. We assume that this time unit is the same for each index. We classically denote by β_t the start-up time, by τ_t the transmission rate in the network and by ω_t the copy rate in the communication buffer. From β_t, τ_t and ω_t, we can deduce the transmission time of an elementary datum. Let us assume that there are currently $M_1, ..., M_n$ messages in transit in the network. When a new message M is sent, we have to take into account its start up time β_t, the current network load $h\tau_t$ and its copy rate in the communication buffer $size(M)\omega_t$. To roughly model adaptative routing and circuit switching, we have: $h = max\{size(M_1), ..., size(M_n), size(M)\}\tau_t$. We obtain the following transmission time:

$$1 + \beta_t + max\{size(M_1), ..., size(M_n), size(M)\}\tau_t + size(M)\omega_t$$

To take into account specificity of architectures based on distant writing in memory, we introduce a blocking time for the execution of a send instruction. Let us assume that there are currently $M_1, ..., M_n$ data in transit in the network. When a message M is sent, the blocking time of the send instruction is:

$$1 + \beta_b + max\{size(M_1), ..., size(M_n), size(M)\}\tau_b + size(M)\omega_b$$

Note that this evaluation of the network load favor the balancing of the communications with regard to the indices.

To take into account the transmission of the clock of the last meaningful instruction, we introduce the γ parameter.

The following array displays the values of these parameters with regard to an elementary instruction (EI) of an implantation of $\mathcal{SCL}$-chan on a Cray T3E.

P (*nb. procs*)	2	4	8	16	32
$EI(10^{-9}sec)$	2.5	2.5	2.5	2.5	2.5
$\beta_t(EI)$	8223	11787	16183	26584	51361
$\tau_t(EI/word)$	0	0	0	0	0
$\omega_t(EI/word)$	59	59	60	65	71
$\beta_b(EI)$	6072	8649	16124	24500	45366
$\tau_b(EI/word)$	0	0	0	0	0
$\omega_b(EI/word)$	34	34	31	41	53
$\gamma(EI)$	8282	11846	16243	26649	51432

Example. The following $\mathcal{SCL}$-chan program displays a sending instruction of an elementary datum X from each index to its right neighbor:

$$\mathsf{send}\ X\ \mathsf{to}\ (This + 1)$$

Let P be the number of indices. We suppose that there is no message in transit at the beginning of the execution. Each index sends one datum message. As there is P elementary messages entering the network at the same time, the corresponding transmission time is $1 + \beta_t + \tau_t + \omega_t$. In the same way, the blocking time for this instruction is $1 + \beta_b + \tau_b + \omega_b$.

3.2 The execution cost

We can model the execution of a $\mathcal{SCL}$-chan program by constructing an array where each column corresponds to an index and each line corresponds to an execution date. This array associates a date to each instance of instruction executed by an index. It corresponds to the number of time units before the execution of this instance by the index. The execution cost for an algorithm is the maximum, for all indices, of the final execution dates. It corresponds to the number of lines in the array.

Example. Let us consider the following shift program.

$$\mathsf{send}\ X\ \mathsf{to}\ (This + 1);$$
$$\mathsf{receive}\ X\ \mathsf{from}\ (This - 1)\ \mathsf{into}\ Y$$

The figure 1 displays the array corresponding to this program. At the beginning of the execution, each index sends its component of the variable X to its neighbor. We suppose that the size of the

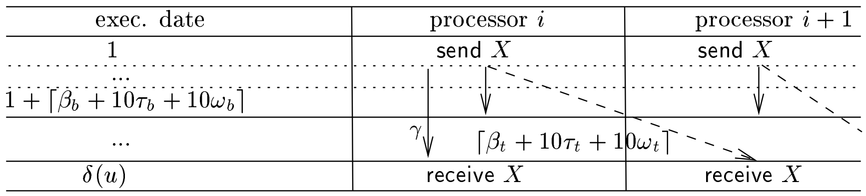


Figure 1: Array for the example 1.

variable X is 10. We denote by $\delta(u)$ the final execution date for the index u and by $\lceil Y \rceil$ the ceiling of the value Y. We obtain:

$$\forall\ u,\ \delta(u) = 1 + max\{1 + \lceil \beta_b + 10\tau_b + 10\omega_b \rceil; 1 + \gamma; 1 + \lceil \beta_t + 10\tau_t + 10\omega_t \rceil\}$$

In a realistic architecture, for a given communication, the blocking time of the sender is not greater than the transmission time of the message. Moreover, a clock transmission can always be modelized by a communication of an elementary message. Therefore, in the following examples, we suppose that for each date, the transmission time is greater than the blocking time and the clock transmission time. Therefore, we can deduce:

$$Cost(S) = max_u\{\delta(u)\} = 1 + \lceil \beta_t + (10P)\tau_t + 10\omega_t \rceil$$

The following array displays this transmission time on a Cray T3E.

P (*nb. procs*)	2	4	8	16	32
$Cost(S)(10^{-6}sec)$	22.03	30.94	41.96	68.09	130.18
$Measured(S)(10^{-6}sec)$	22	28.2	42.5	67	124.5

3.3 A structural complexity function

Our goal is to associate a symbolic cost to each execution of a program S starting with an initial environment σ, on an architecture characterized by the parameters described before. A natural approach is to define a complexity function by induction on the program structure. For a given program S, the complexity function $[\![S]\!]$ is defined from *State* to *State*. A state is composed by an environment σ, the channel state Ch and for each index u and the execution date $\delta(u)$ and the execution date associated to a meaningful instruction $\alpha(u)$. The environment σ contains the value of each variable of the program. Each component of the communication array Ch symbolizes a communication channel. It contains the last datum sent in this channel and the corresponding execution date. We imposes that a receive can only be performed after the execution date stored. Thanks to the component α, a receive can only be performed when the sender has executed the last meaningful instruction appearing before in the program.

Definition 3 (State) *We call state an tuple $<\sigma, Ch, \delta, \alpha>$ where σ is an environment, Ch stores the communication channels. For each index u, $\delta(u)$ is the current execution time and $\alpha(u)$ stores the execution date of the last instance of a meaningful instruction executed by u.*

For a given state $<\sigma, Ch, \delta, \alpha>$, given indices u and v, and a variable A, $\sigma(A)|_u$ corresponds to the current value of the variable A of the index u. We introduce the boolean array $\#$ encoding the current context: $\sigma(\#)|_u$ encodes the activity of the index u. We also introduce an array Net encoding the network activity: $\sigma(Net)|_u$ stores the size of each message sent by u and its corresponding date. The array Net will be used in the second step to compute the transmission time function. $Ch(u, v, A)$ stores the last value sent in the channel (u, v, A) and the corresponding execution date. Since the structural complexity is based on an induction on the program structure, we only have to store the last value in the channel. We denote by $< \sigma, Nil, 0, 0 >$ the state $< \sigma, Ch, \delta, \alpha >$ where for all index u, v and all variable A , we have $Ch(u, v, A) = Nil$, $\delta(u) = 0$, $\alpha(u) = 0$. We denote by $State$ the set of all states.

skip.

$$[\![\mathsf{skip}]\!](< \sigma, Ch, \delta, \alpha >) =< \sigma, Ch, \delta', \alpha >$$

where

$$\forall\ u\ \delta'(u) = \begin{cases} \delta(u) + 1 \text{ if } \sigma(\#)|_u \\ \delta(u) \text{ otherwise.} \end{cases}$$

Assignment.

$$[\![X := E]\!](< \sigma, Ch, \delta, \alpha >) =< \sigma', Ch, \delta', \alpha >$$

where

$$\forall\ u, A\ \sigma'(A)|_u = \begin{cases} \sigma(E)|_u \text{ if } \sigma(\#)|_u \text{ and } A = X, \\ \sigma(A)|_u \text{ otherwise.} \end{cases}$$

$$\forall\ u\ \delta'(u) = \begin{cases} \delta(u) + 1 \text{ if } \sigma(\#)|_u, \\ \delta(u) \text{ otherwise.} \end{cases}$$

A difficulty arises for the receive and send semantics, as the network load cannot be incrementally evaluated following the program structure.

Example. Let us consider the execution of the following program with two indices. At the execution date 2, the index 0 executes the first send instruction while the index 1 executes the second send. The network load corresponding to the first send has to take into account the execution of a send appearing after in the program structure.

```
where This = 0 do
  send X to This + 1
end;
send X to This − 1
```

Therefore, the evaluation of the transmission time is postponed after the computation of the complexity. We define a function $Load(d)$ associating an network load to an execution date d. From this load function, we can deduce the blocking function of a send instruction and the transmission time of a message. We denote by s the size of the message.

$$w(d, s) = d + 1 + \beta_t + Load(d)\tau_t + s\omega_t$$
$$b(d, s) = d + 1 + \beta_b + Load(d)\tau_b + s\omega_b$$

exec. date	processor u	processor v
d	send X	
...		
$\alpha(u)$	meaning. inst.	
...		$w(d, size(Val))$
$\delta(v)$	$\alpha(u+1)$	$+\gamma$ Inst
...		
$\delta'(v)$		receive X

Figure 2: Array for the execution of a receive instruction.

Sending instruction. This transition adds a new message and the current execution date in the variable Ch. The array Net stores the size of the message.

$$[\![\text{send } X \text{ to } Y]\!](<\sigma, Ch, \delta, \alpha>) = <\sigma', Ch', \delta', \alpha'>$$

where

$$\forall\, u,\ A,\ \sigma'(A)|_u = \begin{cases} \{(\delta'(u), size(X))\} \cup \sigma(A)|_u \text{ if } \sigma(\#)|_u \text{ and } A = Net \\ \sigma(A)|_u \text{ otherwise.} \end{cases}$$

$$\forall\, u,\ ,v,\ A\ Ch'(u,v,A) = \begin{cases} (\sigma(X)|_u, \delta(u)) \text{ if } \sigma(\#)|_u,\ v = \sigma(Y)|_u \text{ and } A = X \\ Ch(u,v,A) \text{ otherwise.} \end{cases}$$

$$\forall\, u\ \delta'(u) = \begin{cases} b(\delta(u), size(X)) \text{ if } \sigma(\#)|_u, \\ \delta(u) \text{ otherwise.} \end{cases}$$

$$\forall\, u\ \alpha'(u) = \begin{cases} \delta(u) \text{ if } \sigma(\#)|_u, \\ \alpha(u) \text{ otherwise.} \end{cases}$$

receive **X** from **Y** into **Z.** The function computes the maximum between the date of the receiver, the date of meaningful instruction of the sender and the date stored in the channel plus the communication cost. The figure 2 displays the corresponding array.

$$[\![\text{receive } X \text{ from } Y \text{ into } Z]\!](<\sigma, Ch, \delta, \alpha>) = <\sigma', Ch', \delta', \alpha>$$

$$\forall\, u, A\ \sigma'(A)|_u = \begin{cases} Val \text{ if } \sigma(\#)|_u,\ (Val, d) = Ch(\sigma(Y)|_u, u, A) \text{ and } A = Z, \\ \sigma(A)|_u \text{ otherwise.} \end{cases}$$

$$\forall\, u,\ v,\ A,\ Ch'(u,v,A) = \begin{cases} Nil \text{ if } \sigma(\#)|_v,\ u = \sigma(Y)|_v \text{ and } A = X, \\ Ch(u,v,A) \text{ otherwise.} \end{cases}$$

$$\forall\, u\ \delta'(u) = \begin{cases} max\{\delta(u), \alpha(\sigma(Y)|_u + 1 + \gamma, w(d, size(Val))\} + 1 \\ \quad \text{if } \sigma(\#)|_u \text{ and } (Val, d) = Ch(\sigma(Y)|_u, u, A), \\ \delta(u) \text{ otherwise.} \end{cases}$$

The sequence.

$$[\![S;T]\!](<\sigma, Ch, \delta, \alpha>) = [\![T]\!]([\![S]\!](<\sigma, Ch, \delta, \alpha>))$$

To compute the resulting state of the evaluation of a conditional structure with two branches, we define the operator $\uplus$ from $State$ x $State$ into $State$. This operator fuses together the two final states of the branches. Note that an index entering or exiting from a conditional structure broadcasts its current clock.

Definition 4 (Operator $\uplus$) *Let $<\sigma, Ch, \delta, \alpha>$ and $<\sigma', Ch', \delta', \alpha'>$ be two states such as for all index u, $\sigma(\#)|_u \wedge \sigma'(\#)|_u = 0$, we define the operator $\uplus$ such as: $<\sigma, Ch, \delta, \alpha> \uplus <\sigma', Ch', \delta', \alpha'> = <\sigma'', Ch'', \delta'', \alpha''>$ where*

$$\forall\ u, A\ \sigma''(A)|_u = \begin{cases} \sigma(A)|_u \ \textit{if}\ \sigma(\#)|_u, \\ \sigma'(A)|_u \ \textit{otherwise.} \end{cases}$$

$$\forall\ u,\ v,\ A\ Ch''(u,v,A) = \begin{cases} Ch(u,v,A)\ \textit{if}\ \sigma(\#)|_u\ \textit{and}\ \sigma(\#)|_v, \\ Ch'(u,v,A)\ \textit{if}\ \sigma'(\#)|_u\ \textit{and}\ \sigma'(\#)|_v, \\ Ch(u,v,A)\ \textit{if}\ \sigma(\#)|_u,\ \sigma'(\#)|_v\ \textit{and}\ Ch(u,v,A) \neq Nil, \\ Ch'(u,v,A)\ \textit{if}\ \sigma'(\#)|_u,\ \sigma(\#)|_v\ \textit{and}\ Ch'(u,v,A) \neq Nil, \\ Ch'(u,v,A)\ \textit{if}\ \sigma(\#)|_u,\ \sigma'(\#)|_v\ \textit{and}\ Ch(u,v,A) = Nil, \\ Ch(u,v,A)\ \textit{if}\ \sigma'(\#)|_u,\ \sigma(\#)|_v\ \textit{and}\ Ch'(u,v,A) = Nil. \end{cases}$$

$$\forall\ u\ \delta''(u) = \begin{cases} \delta(u)\ \textit{if}\ \sigma(\#)|_u, \\ \delta'(u)\ \textit{if}\ \sigma'(\#)|_u, \\ \delta(u)\ \textit{otherwise.} \end{cases}$$

$$\forall\ u\ \alpha''(u) = \begin{cases} \delta''(u)\ \textit{if}\ \sigma(\#)|_u\ \textit{or}\ \sigma'(\#)|_u, \\ \alpha(u)\ \textit{otherwise.} \end{cases}$$

The conditional.

$$[\![\mathsf{where}\ B\ \mathsf{do}\ S\ \mathsf{elsewhere}\ Q\ \mathsf{end}]\!](<\sigma, Ch, \delta, \alpha>) = ([\![S]\!](<\sigma', Ch', \delta', \alpha'>)) \uplus ([\![Q]\!](<\sigma'', Ch'', \delta', \alpha'>))$$

where

$$\forall\ u, A\ \sigma'(A)|_u = \begin{cases} \sigma(\#)|_u \text{ if } \sigma(B)|_u \text{ and } A = \#, \\ False \text{ if } \neg\sigma(B)|_u \text{ and } A = \#, \\ \sigma(A)|_u \text{ otherwise.} \end{cases}$$

$$\forall\ u, A\ \sigma''(A)|_u = \begin{cases} \sigma(\#)|_u \text{ if } \neg\sigma(B)|_u \text{ and } A = \#, \\ False \text{ if } \sigma(B)|_u \text{ and } A = \#, \\ \sigma(A)|_u \text{ otherwise.} \end{cases}$$

$$\forall\ u, v, A\ Ch'(u,v,A) = \begin{cases} Nil \text{ if } \sigma'(\#)|_u \text{ and } \neg\sigma'(\#)|_v, \\ Ch(u,v,A) \text{ otherwise.} \end{cases}$$

$$\forall\ u, , v, A\ Ch''(u,v,A) = \begin{cases} Nil \text{ if } \sigma''(\#)|_u \text{ and } \neg\sigma''(\#)|_v, \\ Ch(u,v,A) \text{ otherwise.} \end{cases}$$

$$\forall\ u, A\ \sigma''(A)|_u = \begin{cases} \sigma(\#)|_u \text{ if } \neg\sigma(B)|_u \text{ and } A = \#, \\ False \text{ if } \sigma(B)|_u \text{ and } A = \#, \\ \sigma(A)|_u \text{ otherwise.} \end{cases}$$

$$\forall\ u\ \delta'(u) = \begin{cases} b(\delta(u), 0) \text{ if } \sigma(\#)|_u, \\ \delta(u) \text{ otherwise.} \end{cases}$$

$$\forall\ u\ \alpha'(u) = \begin{cases} \delta(u) \text{ if } \sigma(\#)|_u, \\ \alpha(u) \text{ otherwise.} \end{cases}$$

Note that the evaluation of the condition B by an active index u increases its counter $\delta(u)$.

Example. The following example displays a broadcast from each index to all the others. Each index performs all the sending instructions and then receives data. For the sake of clarity, we define a for loop called $\mathsf{forwhere}\ I := A\ \mathsf{to}\ B$ where A and B are scalar date. We consider such structure as a sequence of all the iterations. Therefore, we do not consider the for construct as a meaningful instruction. We compute the structural complexity. The following array displays all the instruction instances of the program and the corresponding values of δ. We denote by $\delta_i(u)$ the execution time resulting from the execution of the instance i by the index u. In the same way, we denote by $\sigma_i|_u$ the environment resulting from the execution of the instance i by the index u. The first column of each array displays the instance number with regard to the program structure. Initially, we suppose $\sigma(\#) = True$ and for all index u, $\delta_0(u) = \alpha_0(u) = 0$.

$Inst$	$Program\ S$	$\delta_{Inst}(u)$	$\sigma_{Inst}(Net)\|_u$
$2I-1$	forwhere $I := 1$ to $P-1$ do	$\delta_{2I-2}(u)+1$	$\sigma_{2I-2}(Net)\|_u$
$2I$	send X to $(This+I)\ mod\ P$	$b(\delta_{2I-1}(u), size(X))$	$\{(\delta_{2I-1}(u), size(X))\}$ $\cup\sigma_{2I-2}(Net)\|_u$
$2(P-1)+1$	end;	$\delta_{2P-2}(u)$	$\sigma_{2P-2}(Net)\|_u$
$2(P-1)+2I$	forwhere $I := 1$ to $P-1$ do	$\delta_{2(P-1)+2I-1}(u)+1$	$\sigma_{2P-2}(Net)\|_u$
$2(P-1)+2I+1$	receive X from	$max\{\delta_{2(P-1)+2I}(u);$	$\sigma_{2P-2}(Net)\|_u$
	$(This-I)\ mod\ P$ into Y	$\delta_{2P-3}(This-u)+1+\gamma;$	$\sigma_{2P-2}(Net)\|_u$
		$w(\delta_{2I-1}(This-u),$ $size(X))\}+1$	$\sigma_{2P-2}(Net)\|_u$
$4(P-1)+2$	end	$\delta_{4(P-1)+1}$	$\sigma_{2P-2}(Net)\|_u$

We obtain the following complexity:

$$[\![S1]\!](<\sigma, Nil, 0, 0>) = <\sigma_{(4P-1)+2}, Nil, \delta_{(4P-1)+2}, \alpha_{4(P-1)+2}>$$

where $\delta_{4(P-1)+2}(u)$ encodes the final execution date of a given index u. The set $\sigma_{4(P-1)+2}(Net)|_u$ stores informations on each messages sent by u. For each index u, the set $\sigma_{4(P-1)+2}(Net)|_u$ stores the datum size and the execution date of each send.

3.4 The Load evaluation

In order to compute the the blocking time and the transmission time due to communications, we want to associate a network load to an execution date. It corresponds to messages sent before this date and not yet received. Therefore, we have to define an order on dates. Note that execution dates are handled as symbolic dates. Let d and d' be two symbolic dates. We say that d is lower than d', denoted by $d < d'$, if the expression d' is an upperbound of the expression d. Since two expressions may be not comparable, the order on dates is only partial.

Example. If d and d' are arbitrary symbolic dates, we have: $d < max\{d; d'\}$, $max\{d; d'\}$ and $max\{d; 5\}$ are not comparable. Note that the transmission time function and the blocking time function are growing functions. If s is an arbitrary message size, we have: $d < w(d, s)$ and $d < b(d, s)$.

Remark. The ordering on symbolic dates refines the classical causality order on the corresponding instances of instruction executed by an index. But they are not generally equal.

In order to compute the network load for a given date d, we define the function $Load(d)$ thanks to the set Net described in the previous section. The function $Load(d)$ must not take into account the messages sent after d and the messages received before d. The load function $Load(d)$ considers each the message m sent at a date d_m lower or not comparable to d. From this set, it selects each messages m arrived at a date $w(d_m, size(m))$ later or not comparable to the date d.

$$Load(d) = max\{s/(s, d') \in \cup_u \sigma(Net)|_u \ \wedge\ d \not< d' \ \wedge\ w(s, d') \not< d\}$$

Remark. If we consider each computer parameters $(\beta_t, \tau_t, \omega_t, \beta_b, \tau_b, \omega_b, \gamma)$ as null value, We can compute a lower bound for the complexity ($\forall\ d, s\ w(d,s) = d+1,\ b(d,s) = d+1$). It corresponds to a computer performing communications infinitely faster than local computations.

Example. Let turn to the broadcast example. Thanks to the set Net previously computed, we can evaluate the function $Load(d)$. Since there is no conditional structure in the program, for a instance of instruction, each index have the same execution date. Therefore, the sending dates are totally ordered. For each iteration I, we have: $\forall\ u, v,\ \delta_{2(I-1)}(v) < \delta_{2I}(u)$. The network load in the iteration I only has to take into account the messages sent during the previous iterations.

$$Load(\delta_{2I}(u)) = size(X)$$

From the values of $Load(d)$, we can directly deduce the blocking time and the transmission time:

$$\begin{aligned} b(\delta_{2I-1}(u), size(X)) &= I(2 + \beta_b + size(X)(\tau_b + \omega_b) \\ w(\delta_{2I-1}(u), size(X)) &= \delta_{2I-1}(u) + 2 + \beta_t + size(X)(\tau_t + \omega_t) \end{aligned}$$

We introduce these functions in the final execution date $\delta_{4(P-1)+2}(u)$. For sake of conciseness, we denote by $\delta(u)$ this date. it is composed by an overlapping of maximum functions. We can reduce this overlapping by computing all the combinations. We gather them into three groups : the dates encoding the transmission of a message, the propagation of a clock or the local computation. We obtain the following final date:

$$\begin{aligned} &\delta(u) = max\{ \\ &\quad max\{2I + (I-1)(\beta_b + size(X)(\tau_b + \omega_b)) \\ &\qquad +\beta_t + size(X)(\tau_t + \omega_t) + 1 + 2(P-1-I)/\forall I \in 1..(P-1)\} \\ &\quad max\{2(P-1) + (P-2)(\beta_b + size(X)(\tau_b + \omega_b)) + 1 + \gamma + 2(P-1-I)/\forall I \in 1..(P-1)\} \\ &\quad (P-1)(2 + \beta_b + size(X)(\tau_b + \omega_b)) + 2(P-1)\} \end{aligned}$$

As discussed before, for a given message, the blocking time is lower than the transmission one. Therefore, we simplify the date:

$$\begin{aligned} &\delta(u) = 2(P-1) + (P-2)(\beta_b + size(X)(\tau_b + \omega_b)) + max\{ \\ &\quad \beta_t + size(X)(\tau_t + \omega_t) + 1; \gamma + 2P - 3; \beta_b + size(X)(\tau_b + \omega_b) + 2(P-1)\} \end{aligned}$$

From this result, we distinguish three cases, according to the parameters values:

First case. The transmission time of the last message overlap the overhead due to the clock management and the blocking time of the last sending instruction:

$$\begin{aligned} \beta_t + size(X)(\tau_t + \omega_t) &\leq \gamma + 2(P-1) \\ \beta_t + size(X)(\tau_t + \omega_t) &\leq \beta_b + size(X)(\tau_b + \omega_b) + 2P - 5 \end{aligned}$$

We deduce the final date and the cost for the program:

$$Cost(S) = \delta(u) = 2P - 5 + (P-2)(\beta_b + size(X)(\tau_b + \omega_b)) + \beta_t + size(X)(\tau_t + \omega_t)$$

We have $Cost(S) = O(size(X)P)$.

Second case. The clock overhead on the first receive overlaps the transmission time and the blocking one:

$$\begin{array}{rcl}\gamma & \leq & \beta_t + size(X)(\tau_t + \omega_t) - 2P + 4 \\ \gamma & \leq & \beta_b + size(X)(\tau_b + \omega_b) - 1\end{array}$$

We deduce the final date and the cost for the program:

$$Cost(S) = \delta(u) = 4P - 3 + (P-2)(\beta_b + size(X)(\tau_b + \omega_b)) + \gamma$$

We have $Cost(S) = O(size(X)P)$.

Third case. The overhead due to the blocking time of the last sending instruction overlaps the transmission time and the overhead due to clock management:

$$\begin{array}{rcl}\beta_b + size(X)(\tau_b + \omega_b) & \leq & \beta_t + size(X)(\tau_t + \omega_t) - 2(P-2) \\ \beta_b + size(X)(\tau_b + \omega_b) & \leq & \gamma + 1\end{array}$$

We deduce the final date and the cost for the program:

$$Cost(S) = \delta(u) = (P-1)(4 + \beta_b + size(X)(\tau_b + \omega_b))$$

We have $Cost(S) = O(size(X)P)$.

The following array displays the estimated result and the experimental one on a Cray T3E where $size(X) = 100$. Note that the estimated cost corresponds to the first case.

P (nb. procs)	2	4	8	16	32
$Cost(S)(10^{-6}sec)$	35	104	344	1082	3944
$Measured(S)(10^{-6}sec)$	41	119	368	1218	4348

4 The scan algorithm

We now study a scan algorithm. Let us denote by P the number of processors and N the number of initial data. Data are distributed by block where $M = N/P$ is the size of each block. This algorithm can be divided into three parts: First the computation of the partial sum on local data, then the computation of the scan and finally the addition of the result of the scan in the local result. For sake of simplicity, we do not compute the channel state and the resulting memory. We only compute the resulting execution dates.

4.1 The first part: The program S_1

$$\left.\begin{array}{l}\textsf{forwhere } I := 2 \textsf{ to } M \textsf{ do} \\ \quad X[I] := X[I] + X[I-1] \\ \textsf{end;}\end{array}\right\} S_1$$

Since the first part is only composed by local computations, its structural complexity does not depend on the transmission time function. We obtain the following result:

$$[\![S_1]\!](<\sigma_0, Ch_0, 0, 0>) = <\sigma, Ch, \delta, \alpha>$$

where $\delta(u) = 2(M-1)$.

4.2 The second part: The program S_2

Inst.	*Program* S_2	$\delta_{Inst}(u)$	$\sigma_{Inst}(Net)\|_u$
1	$Y := X[M]$;	$\delta_{S1}(u)+1$	$\sigma_{S1}(Net)\|_u$
2	send Y to $This+1$;	$b(\delta_{S1}(u)+1, size(Y))$	$\{\delta_{S1}(u)+1; size(Y)\}$ $\cup\sigma_2(Net)\|_u$
$4I-1$	forwhere $I := 1$ to $\lfloor log_2(This)\rfloor$ do	$\delta_{4I-2}(u)+1$	$\sigma_{4I-2}(Net)\|_u$
$4I$	receive Y from $This - 2^{(I-1)}$ into Z;	$max\{\delta_{4I-1}(u);$ $\delta_{4I-3}(u-2^{(I-1)})+1+\gamma;$ $w(\delta_{4I-3}(u-2^{(I-1)}), size(Y))\}$	$\sigma_{4I-2}(Net)\|_u$
$4I+1$	$Y := Y+Z$;	$\delta_{4I}(u)+1$	$\sigma_{4I-2}(Net)\|_u$
$4I+2$	send Y to $This+2^I$;	$b(\delta_{4I+1}(u), size(Y))$	$\{\delta_{4I+1}(u)+1; size(Y)\}$ $\cup\sigma_{4I-2}(Net)\|_u$
$4\lfloor log_2(u)\rfloor+3$	end;	$\delta_{4\lfloor log_2(u)\rfloor+2}(u)$	$\sigma_{4\lfloor log_2(u)\rfloor+2}(Net)\|_u$

In S_2, communications occur. Therefore, the structural complexity depends on the transmission time and the blocking time. We first compute the load function. Since we have $\delta_{4I-2}(u-2^{(I-1)}) < \delta_{4I+3}(u)$, the network load of a given iteration only has to take into account the messages sent during the previous iteration. We obtain:

$$Load(2(M-1)+1) = size(Y)$$
$$Load(\delta_{4I+2}(u)) = size(Y)$$

For each receive instruction, the corresponding send is the last meaningful instruction appearing before it. Therefore, the structural clock and the communication message are sent at the same execution date. As discussed in the previous example, we do not have to take into account the transmission of the clock. We compute the execution date by induction on the iterations of the forwhere loop. If we denote the complexity as follows:

$$[\![S_2]\!](<\sigma, Ch, \delta, \alpha>) = <\sigma', Ch', \delta', \alpha'>$$

then we obtain:

$$\begin{aligned}\delta'(u) &= \delta_{4\lfloor log_2(u)\rfloor+3}(u)\\ &= 2M + log_2(u)[2+\beta_t+size(Y)(\tau_t+\omega_t)] + 1 + \beta_b + size(Y)(\tau_b+\omega_b)\end{aligned}$$

4.3 The third part: The program S_3

$$\left.\begin{array}{l}\text{forwhere } I := 1 \text{ to } M \text{ do}\\ \quad X[I] := X[I] + Y - X[M]\\ \text{end}\end{array}\right\} S_3$$

Since the third part is only composed by local computation, its structural complexity does not depend on the transmission time function. We obtain:

$$[\![S_3]\!](<\sigma', Ch', \delta', \alpha'>) = <\sigma'', Ch'', \delta'', \alpha''>$$

where $\delta''(u) = \delta'(u) + 2M$.

4.4 The final result

We now compute the structural complexity for the scan algorithm:

$$[\![P]\!](<\sigma_0, Ch_0, 0, 0>) = <\sigma'', Ch'', \delta'', \alpha''>$$

where

$$\delta''(u) = 4M + log_2(u)[2 + \beta_t + size(Y)(\tau_t + \omega_t)] + 1 + \beta_b + size(Y)(\tau_b + \omega_b)$$

From the complexity function, we deduce:

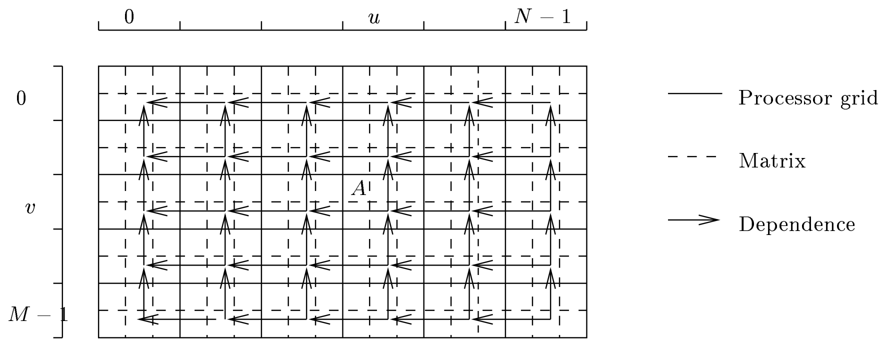
$$Cost(S) = 4N/P + log_2(P)[1 + \beta_t + size(Y)(\tau_t + \omega_t)] + \beta_b + size(Y)(\tau_b + \omega_b)$$

The following array displays both measured and estimated results on a Cray T3E for 32 processors. We suppose that $size(Y) = 1\ word$.

N ($word$)	64	320	3200	32000
$Cost(S)$ ($10^{-6}sec.$)	756	756	757	759
$Measured(S)$ ($10^{-6}sec.$)	694	701	718	792

5 The wave

Our next example is inspired by the Gauss-Seidel relaxation algorithm. It consists in going through a two dimensional matrix G from right to left and top to bottom, while computing the sum of its upper and left neighbors. It is an interesting problem since it includes numerous dependencies. Each index only owns a sub-matrix. We denotes by (I, J) the index owning the sub-matrix $G(I, J) = A$. The size of the processor grid is $NM = P$.



The submatrix computed by the left and upper neighbors are respectively stored in the submatrix $Tmp1$ and $Tmp2$. We respectively denote by $column(A)$, $line(A)$ and $size(A)$, the number of column, line and the size of the matrix A. The procedure $Local(A)$ computes of the relaxation on the sub-matrix A.

$Local(A)$
$A[1,1] := Tmp1[column(A),1] + Tmp2[1,line(A)]$;
for $X = 2$ to $column(A)$ do
$A[X,1] := A[X-1,1] + Tmp2[X,line(A)]$
end;
for $Y = 2$ to $line(A)$ do
$A[1,Y] := Tmp1[column(A),Y] + A[1,Y-1]$
end;
for $X = 2$ to $column(A)$ do
for $Y = 2$ to $line(A)$ do
$A[X,Y] := A[X-1,Y] + A[X,Y-1]$
end
end

When the complexity computes the execution date of an index for this procedure, it increases the current date by $3size(A)$.

5.1 The translated algorithm

The following parallel algorithm relies on a simple and intuitive translation of the sequential program in an $\mathcal{SCL}$-chan one.

$Inst$	$Program\ S$
$(9M+2)I$	for $I = 0$ to $N-1$ do
$(9M+2)I+9J+1$	for $J = 0$ to $M-1$ do
$(9M+2)I+9J+2$	where $This = (I,J)$ do
$(9M+2)I+9J+3$	receive A from $(I-1,J)$ into $Tmp1$;
$(9M+2)I+9J+4$	receive A from $(I,J-1)$ into $Tmp2$;
$(9M+2)I+9J+5$	$Local(A)$;
$(9M+2)I+9J+6$	send A to $(I+1,J)$;
$(9M+2)I+9J+7$	send A to $(I,J+1)$
$(9M+2)I+9J+8$	end
$(9M+2)I+9(J+1)$	end
$(9M+2)I+9M+1$	end

5.1.1 First step: construction of the symbolic dates

$Inst$	$\delta_{Inst}(u,v)$	$\sigma_{inst}(Net)\|_{(u,v)}$
$(9M+2)I$	$\delta_{(9M+2)I-1}(u,v)+1$	$\sigma_{(9M+2)I-1}(Net)\|_{(u,v)}$
$(9M+2)I+9J+1$	$\delta_{(9M+2)I+9J}(u,v)+1$	$\sigma_{(9M+2)I+9J}(Net)\|_{(u,v)}$
$(9M+2)I+9J+2$	$\delta_{(9M+2)I+9J+1}(u,v)+1$	$\sigma_{(9M+2)I+9J}(Net)\|_{(u,v)}$
$(9M+2)I+9J+3$	$[This=(I,J)?max\{\delta_{(9M+2)I+9J+2}(u,v);$ $\delta_{(9M+2)I+9J}(I-1,J)+1+\gamma;$ $w(\delta_{(9M+2)(I-1)+9J+6}(I-1,J),size(A))\}+1]$	$\sigma_{(9M+2)I+9J}(Net)\|_{(u,v)}$
$(9M+2)I+9J+4$	$[This=(I,J)?max\{\delta_{(9M+2)I+9J+3}(u,v);$ $\delta_{(9M+2)I+9J}(I,J-1)+\gamma;$ $w(\delta_{(9M+2)I+9(J-1)+5}(I,J-1),size(A))\}+1]$	$\sigma_{(9M+2)I+9J}(Net)\|_{(u,v)}$
$(9M+2)I+9J+5$	$[This=(I,J)?\delta_{(9M+2)I+9J+4}(u,v)+3size(A)]$	$\sigma_{(9M+2)I+9J}(Net)\|_{(u,v)}$
$(9M+2)I+9J+6$	$[This=(I,J)?$ $b(\delta_{(9M+2)I+9J+5}(u,v),size(A))]$	$(\delta_{(9M+2)I+9J+5}(u,v),size(A))$ $\cup\sigma_{(9M+2)I+9J}(Net)\|_{(u,v)}$
$(9M+2)I+9J+7$	$[This=(I,J)?$ $b(\delta_{(9M+2)I+9J+6}(u,v),size(A))]$	$(\delta_{(9M+2)I+9J+6}(u,v),size(A))$ $\cup\sigma_{(9M+2)I+9J+6}(Net)\|_{(u,v)}$
$(9M+2)I+9J+8$	$\delta_{(9M+2)I+9J+7}(u,v)$	$\sigma_{(9M+2)I+9J+7}(Net)\|_{(u,v)}$
$(9M+2)I+9(J+1)$	$\delta_{(9M+2)I+9J+8}(u,v)$	$\sigma_{(9M+2)I+9J+8}(Net)\|_{(u,v)}$
$(9M+2)I+9M+1$	$\delta_{(9M+2)I+9M}(u,v)$	$\sigma_{(9M+2)I+9M}(Net)\|_{(u,v)}$

For sake of conciseness, we denote by $\delta(u,v)$ the final execution date $\delta_{N(9M+2)-1}(u,v)$ of each index (u,v). We compute the date $\delta(u,v)$ by double induction on u and v. This induction follows the data dependencies:

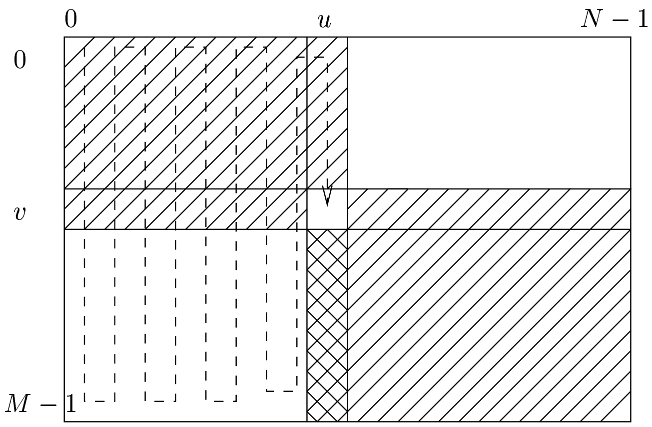
$$\begin{aligned}
&\delta(u,v) \ = \\
&\quad b(b(max\{\delta_0(u,v)+u(2M+1)+2v+5; \\
&\qquad \delta_{(9M+2)u+9v}(u-1,v)+\gamma+3; \delta_{(9M+2)u+9v}(u,v-1)+\gamma+2; \\
&\qquad w(\delta_{(9M+2)(u-1)+9v+6}(u-1,v),size(A))+2; w(\delta_{(9M+2)u+9(v-1)+7}(u,v-1),size(A))+1\} \\
&\quad +3size(A),size(A)),size(A))+(2M+1)(N-u-1)+2(M-v-1)
\end{aligned}$$

remark. Note that we can construct the set Net by finding all the function $b()$ in the execution date.

5.1.2 Second step: evaluation of the network load

We want to associate a network load to each date $\delta_{(9M+2)u+9v+6}(u,v)$ and $\delta_{(9M+2)u+9v+7}(u,v)$ corresponding to the execution of a send.

Since we want to evaluate the network load, we have to instantiate the initial execution date. We suppose that $\forall\ (u,v), \delta_0(u,v)=0$.



- We focus on the messages sent by an index (u', v') such as $u' \leq u$ and $v' \leq v$. We have:

$$\begin{array}{l} w(\delta_{(9M+2)u'+9v'+6}(u', v')) \leq \delta_{(9M+2)u+9v+6}(u, v) < \delta_{(9M+2)u+9v+7}(u, v) \\ w(\delta_{(9M+2)u'+9v'+7}(u', v')) \leq \delta_{(9M+2)u+9v+6}(u, v) < \delta_{(9M+2)u+9v+7}(u, v) \end{array}$$

  Therefore, these messages are not taken into account for the network load estimation.

- We focus on the messages sent by an index (u', v') such as $u' \geq u$ and $v' > v$. We have:

$$\begin{array}{l} w(\delta_{(9M+2)u+9v+6}(u, v)) < \delta_{(9M+2)u'+9v'+6}(u', v') \\ w(\delta_{(9M+2)u+9v+7}(u, v)) < \delta_{(9M+2)u'+9v'+6}(u', v') \\ w(\delta_{(9M+2)u+9v+6}(u, v)) < \delta_{(9M+2)u'+9v'+7}(u', v') \\ w(\delta_{(9M+2)u+9v+7}(u, v)) < \delta_{(9M+2)u'+9v'+7}(u', v') \end{array}$$

  Therefore, these messages are not taken into account too.

- We focus on the messages sent by an index (u', v') such as $u' \geq u$ and $v' = v$. We have:

$$\begin{array}{l} w(\delta_{(9M+2)u+9v+6}(u, v)) < \delta_{(9M+2)u'+9v'+6}(u', v') \\ w(\delta_{(9M+2)u+9v+7}(u, v)) < \delta_{(9M+2)u'+9v'+7}(u', v') \end{array}$$

  Therefore, these messages are not taken into account for a load estimation of date $\delta_{(9M+2)u+9v+6}(u, v)$.

We can deduce the following network load:

$$\begin{array}{l} Load(\delta_{(9M+2)u+9v+6}(u, v)) = size(A) \\ Load(\delta_{(9M+2)u+9v+7}(u, v)) = size(A) \end{array}$$

5.1.3 The final result

Next, we introduce the value of the network load in the execution date $\delta(u, v)$.

$$\begin{array}{l} \delta(u,v) = \quad max\{u(2M+1) + 2v + 5; \\ \quad \delta(u-1,v) - [(2M+1)(N-u-1) + 2(M-v-1)] + \gamma + 3; \\ \quad \delta(u,v-1) - [(2M+1)(N-u-1) + 2(M-v-1)] + \gamma + 2; \\ \quad \delta(u-1,v) - [(2M+1)(N-u) + 2(M-v-1)] \\ \quad\quad -2(1+\beta_b + size(A)(\tau_b+\omega_b)) + 1 + \beta_t + size(A)(\tau_t+\omega_t); \\ \quad \delta(u,v-1) - [(2M+1)(N-u-1) + 2(M-v)] \\ \quad\quad -(1+\beta_b + size(A)(\tau_b+\omega_b)) + 1 + \beta_t + size(A)(\tau_t+\omega_t)\} \\ \; +3size(A) + 2(1+\beta_b+size(A)(\tau_b+\omega_b)) + (2M+1)(N-u-1) + 2(M-v-1) \end{array}$$

We simplify this expression:

$$\begin{aligned}\delta(u,v) = \quad & max\{(2M+1)N+4+2(\beta_b+size(A)(\tau_b+\omega_b));\\ & \delta(u-1,v)+\gamma+5+2(\beta_b+size(A)(\tau_b+\omega_b));\\ & \delta(u,v-1)+\gamma+4+2(\beta_b+size(A)(\tau_b+\omega_b));\\ & \delta(u-1,v)-2M+\beta_t+size(A)(\tau_t+\omega_t);\\ & \delta(u,v-1)+\beta_b+size(A)(\tau_b+\omega_b)+\beta_t+size(A)(\tau_t+\omega_t)\}\\ & +3size(A)\end{aligned}$$

From this result, we deduce: $\delta(u,v) \le \delta(N-1,M-1)$. Therefore, we have: $Cost(S) = \delta(N-1,M-1)$

We can now distinguish three cases, according to the parameters values:

First case. Each communication between an index and its neighbor overlaps the corresponding waiting mechanism:

$$\gamma+2M+5+2(\beta_b+size(A)(\tau_b+\omega_b)) < \beta_t+size(A)(\tau_t+\omega_t)$$

Then, we only have to take into account the transmission times of data. We obtain:

$$Cost(S) = 4+N+(N+M-2)[\beta_t+size(A)(\tau_t+\omega_t)+3size(A)]+(M+2)[\beta_b+size(A)(\tau_b+\omega_b)]$$

We see that $Cost(S) = O((M+N)size(A))$.

Second case. The communication between an index and its upper neighbor overlaps the corresponding waiting mechanism between them. But the communication between it and its left neighbor does not overlap the waiting mechanism:

$$\gamma+4+(\beta_b+size(A)(\tau_b+\omega_b)) < \beta_t+size(A)(\tau_t+\omega_t) \le \gamma+2M+5+2(\beta_b+size(A)(\tau_b+\omega_b))$$

In this case, we only have to consider the message transmission time between the columns and the clock transmission time between the lines. We have then:

$$\begin{aligned}Cost(S) = \quad & (2M+4)N+4+(2+2N+M)[\beta_b+size(A)(\tau_b+\omega_b)]+(N-1)\gamma\\ & +(M-1)[\beta_t+size(A)(\tau_t+\omega_t)]+3(N+M)size(A)\end{aligned}$$

We have: $Cost(S) = O(NM+(M+N)size(A))$.

Since the clock mechanism overlaps communication of data, we count the empty iterations.

Third case. The communications between an index and its neighbors never overlaps the waiting mechanism. Therefore, we have the following constraint on the parameters:

$$\beta_t+size(A)(\tau_t+\omega_t) \le \gamma+4+\beta_b+size(A)(\tau_b+\omega_b)$$

In this case, we only have to consider the clock transmission. We have then :

$$Cost(S) = 2MN+6N+4M+2+(\gamma+3size(A))(N+M-2)+(1+N+M)[2(\beta_b+size(A)(\tau_b+\omega_b))]$$

As for the previous case, we have: $Cost(S) = O(NM+(M+N)size(A))$.

Since the clock management overlaps all the communications, the cost does not depend on transmission parameters $(\beta_t,\ \tau_t,\ \omega_t)$.

5.2 A better algorithm

Since the previous algorithm distributes each loop iteration on all the indices, an index executes empty iterations which involves a cost in $O(NM)$ for some architectures. It is often possible to detect at compile time such iterations thanks to a static analysis [14].

We say an iteration is *empty* for an index if this index has nothing else to do but performing tests and handling clocks. Therefore, an index can jump over empty iterations if it increases its structural clock from the number of jumped iterations.

We introduce a new loop structure called *jump* loop. This loop retains the semantics of the forwhere loop but each index u only performs iterations which might be *useful*. Therefore, in order to ensure structural clocks coherence, such index has to update its local clock from the number of meaningful instructions in the skipped iterations.

Inst	*Program* S'
1	jump $(I,J) = This$ do
2	receive A from $(I-1,J)$ into $Tmp1$;
3	receive A from $(I,J-1)$ into $Tmp2$;
4	$Local(A)$;
5	send A to $(I+1,J)$;
6	send A to $(I,J+1)$
7	end

The evaluation of the symbolic dates (in the first step) for the previous algorithm can be reused. We obtain the following complexity:

$$[\![S']\!](<\sigma_0, Ch_0, 0, 0>) = <\sigma, Ch, \delta_7, \alpha>$$

where

$$\begin{aligned}
&\delta_7(u,v) = \\
&\quad b(b(max\{2; \\
&\qquad \delta_7(u-1,v)+\gamma+3; \delta_7(u,v-1)+\gamma+2; \\
&\qquad w(\delta_5(u-1,v), size(A))+2; w(\delta_6(u,v-1), size(A))+1\} + 3size(A)))
\end{aligned}$$

We obtain the same network load as in the previous algorithm:

$$\forall\ (u,v)\ Load(\delta_5(u,v)) = Load(\delta_6(u,v)) = size(A)$$

For sake of clarity, we denote by $\delta(u,v)$ the execution date $\delta_7(u,v)$. As in the previous section, we compute the final execution date thanks to a induction on the indices:

$$\begin{aligned}
&\forall (u,v), \delta(u,v) = \\
&\quad max\{4+2[\beta_b + size(A)(\tau_b+\omega_b)]; \\
&\qquad \delta(u-1,v)+\gamma+5+2[\beta_b+size(A)(\tau_b+\omega_b)]; \\
&\qquad \delta(u,v-1)+\gamma+4+2[\beta_b+size(A)(\tau_b+\omega_b)]; \\
&\qquad \delta(u-1,v)+\beta_b+size(A)(\tau_b+\omega_b)+\beta_t+size(A)(\tau_t+\omega_t)+4; \\
&\qquad \delta(u,v-1)+2[\beta_b+size(A)(\tau_b+\omega_b)]+\beta_t+size(A)(\tau_t+\omega_t)+3\} \\
&\quad +3size(A)
\end{aligned}$$

We obtain: $Cost(S') = \delta(N-1, M-1)$

As discussed before, we have $\beta_t + size(A)(\tau_t+\omega_t) > \gamma$ (the transmission of a datum overlaps the clock transmission). We can now distinguish two cases, according to the parameters values:

First case. Each communication between an index and its neighbor overlaps the corresponding waiting mechanism of $\mathcal{SCL}$-chan between them:

$$\gamma + 1 + \beta_b + size(A)(\tau_b + \omega_b) < \beta_t + size(A)(\tau_t + \omega_t)$$

We only have to take into account the transmission times of data. We obtain then:

$$\begin{aligned} Cost(S') = \quad & 2 + 4N + 4M + (2 + N + 2M)[\beta_b + size(A)(\tau_b + \omega_b)] \\ & +(N + M - 2)[\beta_t + size(A)(\tau_t + \omega_t) + 3size(A)] \end{aligned}$$

We see that $Cost(S') = O((M + N)size(A))$.

Second case. The communication between an index and its upper neighbor overlaps the corresponding waiting mechanism between them. But the communication between it and its left neighbor does not overlap the waiting mechanism. Therefore, we have the following constraint on the parameters:

$$\gamma + 1 + \beta_b + size(A)(\tau_b + \omega_b) \geq \beta_t + size(A)(\tau_t + \omega_t)$$

In this case, we only have to consider the message transmission time between the columns and the clock transmission time between the lines. We have then:

$$\begin{aligned} Cost(S') = \quad & 2 + 5N + 4M + (1 + M + N)2[\beta_b + size(A)(\tau_b + \omega_b)] \\ & +(M - 1)[\beta_t + size(A)(\tau_t + \omega_t)] + (N - 1)\gamma + (N + M)3size(A)] \end{aligned}$$

As for the previous case, we obtain: $Cost(S') = O((M + N)size(A))$.

The jump control structure suppresses the overhead caused by useless iterations.

Remark. In contrast to classical techniques, such as the polytope model [5], the $\mathcal{SCL}$-chan program can exploit the parallelism of a wave algorithm without performing a modification of the instructions ordering. The $\mathcal{SCL}$-chan model appears to be an interesting target language for an efficient automatic translation of sequential arbitrary DOACROSS loops.

5.3 The data mapping

We study now two different data mappings by modifying the values of N, M and the matrix A.

Line mapping. An index owns an entire line of the matrix. Therefore, the number of column of the index grid is $N = 1$. We distinguish the same two cases as described in the previous section:

$$Cost(S'_l) = 6 + 4M + (3 + 2M)[\beta_b + size(A)(\tau_b + \omega_b)] + (M - 1)[\beta_t + size(A)(\tau_t + \omega_t) + 3size(A)]$$

We deduce that $Cost(S'_l) = O(M(size(A)))$.

Column mapping. An index owns an entire column of the matrix. Therefore, the number of line of the index grid is $M = 1$. We distinguish the same two cases as described in the previous section:

First case. Each communication between an index and its neighbor overlaps the corresponding waiting mechanism of $\mathcal{SCL}$-chan between them. Therefore, we have the following constraint on the parameters:

$$\gamma + 1 + \beta_b + size(A)(\tau_b + \omega_b) < \beta_t + size(A)(\tau_t + \omega_t)$$

We only have to take into account the data transmissions time. We obtain then:

$$Cost(S'_c) = 5+4N+(3+N)[\beta_b+size(A)(\tau_b+\omega_b)]+(N-1)[\beta_t+size(A)(\tau_t+\omega_t)+3size(A)]$$

We deduce that $Cost(S'_c) = O(M(size(A)))$.

Second case. The communication between it and its left neighbor does not overlap the waiting mechanism. Therefore, we have the following constraint on the parameters:

$$\gamma + 1 + \beta_b + size(A)(\tau_b + \omega_b) \geq \beta_t + size(A)(\tau_t + \omega_t)$$

In this case, we only have to consider the clock transmission time between the lines. We have then:

$$Cost(S'_c) = 5 + 5N + (3 + 2N)[\beta_b + size(A)(\tau_b + \omega_b)] + (N - 1)[\gamma + 3size(A)]$$

As for the previous case, we obtain: $Cost(S'_c) = O(N(size(A)))$.

We want to compare these results. Suppose that the matrix is a squared one. Therefore, we have: $size(A_c) = size(A_l)$. As discussed before, we suppose that we have $\beta_t + size(A)(\tau_t + \omega_t) \geq \gamma$. The column mapping is better than the line one. The difference between them is due to the ordering on the send an receive instruction in the program.

6 Worst case cost

In this section, we want to cut ourselves off from the environment to compute a worst case cost for a given algorithm.

An asynchronous execution of a parallel program can be viewed as a Direct Acyclic Graph (DAG); where a node represents the execution of an instance of instruction by a single processor and an arrow symbolizes a sequence. The order defined by this DAG has to respect the causality order. it can be total (as in the PRAM model) or partial (as in the BSP and LogP model). The weight on arrows depends on the modelization of the architecture. The cost associated to an execution is then the longest path in the graph.

To compute a worst case cost, one can think of two different methods.

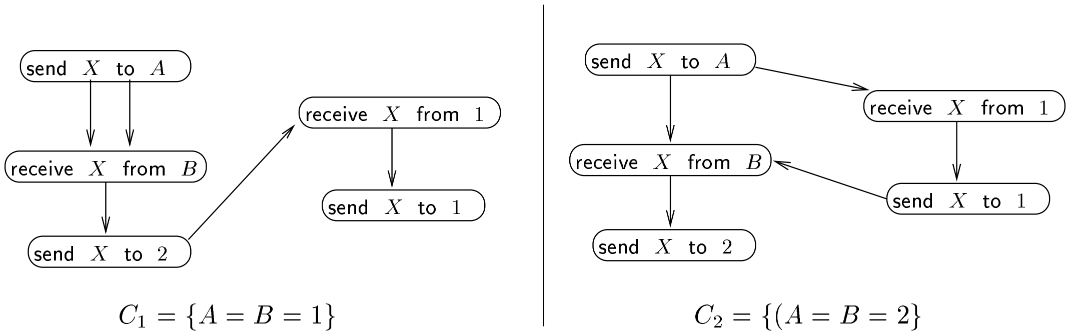
6.1 A graph gathering

The first method consists in considering the algorithm DAG as the gathering of all possible DAGs.

Example. Let S be the following MPI-like program.

$$S \left(\begin{array}{l} \textsf{send } X \textsf{ to } A; \\ \textsf{receive } X \textsf{ from } B; \\ \textsf{send } X \textsf{ to } 2 \end{array} \right) \| \left(\begin{array}{l} \textsf{receive } X \textsf{ from } 1; \\ \textsf{send } X \textsf{ to } 1 \end{array} \right)$$

Let us consider the two executions E_1 and E_2 corresponding respectively to initial conditions $C_1 = \{A = B = 1\}$ and $C_2 = \{A = B = 2\}$. For sake of conciseness, we do not take into account the communication overhead and we suppose that all instructions have the same elementary execution time. We obtain the following two-parts DAG:



$C_1 = \{A = B = 1\}$ $C_2 = \{(A = B = 2\}$

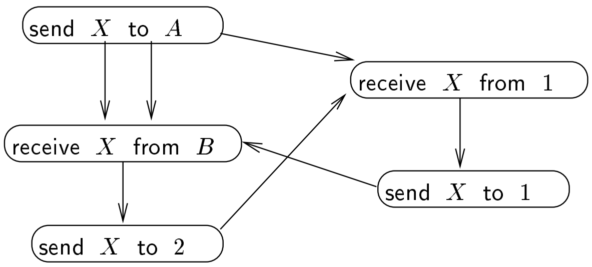
$$C = \{A = B \wedge A \in \{1, 2\}\}$$

We have: $Cost(E_1) = 5$ and $Cost(E_2) = 4$. The resulting worst case cost for the initial condition $C = \{A = B \wedge A \in \{1, 2\}\}$ is $Cost(E) = 5$.

6.2 A structural graph

Gathering the DAG of each execution can lead to a combinatorial explosion. To limit the number of node of the DAG, we only consider one node by instance of instructions. We construct then a graph that subsumes all the execution DAGs by adding sequence arrows corresponding to more possible dependencies. With classical message passing models, appending arrows can result in cycles, thereby forbidding the computation of a longest path.

Example. Let turn to the previous example. If we consider the initial condition $C = \{A = B \wedge A \in \{1, 2\}\}$, we construct the following graph:



$$C = \{A = B \wedge A \in \{1, 2\}\}$$

The graph has to take into account all the possible values of the variable A and B corresponding to a given execution of the program with a different communication scheme. Therefore to compute the corresponding cost, we have to add a arrow which introduce a cycle in the DAG. Therefore, we cannot deduce a cost from such graph.

In the $\mathcal{SCL}$-chan model, the structured communications avoid to introduce cycle in the DAG.

Example: Let us consider the corresponding $\mathcal{SCL}$-chan program. For sake of conciseness, we only display the execution date of each index and we suppose that X is an elementary datum.

$Inst$	$Program\ S$	$\delta_{Inst}(u)$
1	where $This = 1$ do	$\delta_0(u)+1$
2	send X to A	$[u=1?B(\delta_1(u),1)]$
3	end;	$\delta_2(u)$
4	where $This = 2$ do	$\delta_2(u)+1$
5	receive X from 1 into Tmp;	$[u=2?max\{\delta_4(u);\delta_4(1)+\gamma;[A=2?w(\delta_1(1),1)]\}+1]$
6	send X to 1	$[u=2?B(\delta_5(u))+1]$
7	end;	$\delta_6(u)$
8	where $This = 1$ do	$\delta_6(u)+1$
9	receive X from B into Tmp;	$[u=1?max\{\delta_8(u);\delta_8(\sigma(B)\|_u)+\gamma;w(\delta_5(\sigma(B)\|_u))\}+1]$
10	send X to 2	$[u=2?B(\delta_9(u),1)]$
11	end	$\delta_{10}(u)$

If we consider $\delta_0(u)=0$, we compute the following network load:

$$Load(\delta_1(1)) = Load(\delta_5(2)) = Load(\delta_{10}(1)) = size(X) = 1$$

We denote by δ the final execution date δ_{11}.

$$\begin{aligned}\delta(1) &= max\{4+\beta_b+\tau_b+\omega_b;[B=1?4+\beta_b+\tau_b+\omega_b:\delta(2)]+\gamma;\\ &\quad [B=1\wedge A=1?2+\beta_t+\tau_t+\omega_t:[B=2?\delta(2)-1-\beta_b-\tau_b-\omega_b+\beta_t+\tau_t+\omega_t]]\}\\ &\quad +2+\beta_b+\tau_b+\omega_b\\ \delta(2) &= max\{3+\beta_b+\tau_b+\omega_b+\gamma;[A=2?2+\beta_t+\tau_t+\omega_t]\}+3+\beta_b+\tau_b+\omega_b\end{aligned}$$

We can compute a worst case complexity by considering each conditional expression as a "max" expression. Note that this worst case expression may not correspond to a given execution.

$$\begin{aligned}\delta(1) &= max\{4+\beta_b+\tau_b+\omega_b+\gamma;\delta(2)+\gamma;\\ &\quad 2+\beta_t+\tau_t+\omega_t;\delta(2)-1-\beta_b-\tau_b-\omega_b+\beta_t+\tau_t+\omega_t\}\\ &\quad +2+\beta_b+\tau_b+\omega_b\\ \delta(2) &= max\{5+\beta_b+\tau_b+\omega_b+\gamma;4+\beta_t+\tau_t+\omega_t\}+1+\beta_b+\tau_b+\omega_b\end{aligned}$$

We deduce:

$$\begin{aligned}\delta(1) &= max\{\delta(2)+\gamma+1+\beta_b+\tau_b+\omega_b;\\ &\quad \delta(2)+\beta_t+\tau_t+\omega_t\}\\ \delta(2) &= max\{5+\beta_b+\tau_b+\omega_b+\gamma;4+\beta_t+\tau_t+\omega_t\}+2+\beta_b+\tau_b+\omega_b\end{aligned}$$

We have: $Cost(S)=\delta(1)$. Two different cases can occur with regard to the architecture parameters:

First case. Each communication between the two indices overlaps the corresponding waiting mechanism:

$$\gamma+1+\beta_b+\tau_b+\omega_b < \beta_t+\tau_t+\omega_t$$

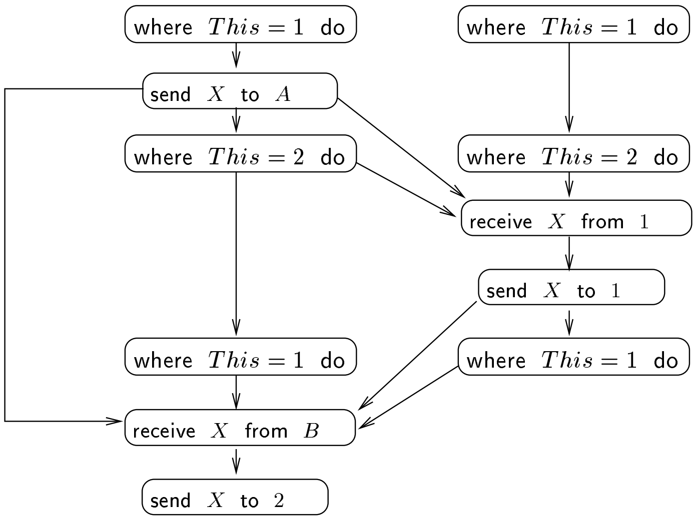
We obtain:

$$Cost(S) = 6+2(\beta_t+\tau_t+\omega_t)+\beta_b+\tau_b+\omega_b$$

Second case. Otherwise, we obtain:

$$Cost(S) = 8 + 3(\beta_b + \tau_b + \omega_b) + 2\gamma$$

If we do not consider the communication overhead, the worst case cost is: $Cost(S) = 8$. This cost corresponds to the longest path in the following structural graph.



$$C = \{A = B \wedge A \in \{1, 2\}\}$$

7 Comparison with other approaches

Valiant and Mc Coll [17, 11] propose an approach called BSP. A computation is a sequence of supersteps. Inside of a superstep, each processor performs local computations and sends messages. Each superstep is completed by a global synchronization to validate the communications for the next superstep. The cost associated to each superstep takes into account local computations, communications and the global synchronization. Thanks to the global synchronization, the communication overhead is globally considered for each superstep. The communication overhead for a given superstep only depends on the computer characteristics and on the number of messages sent in the superstep. Therefore, The BSP model is a simple programming and cost model. As the $\mathcal{SCL}$-chan cost model, it yields a structured communication scheme thanks to the superstep. It allows to construct a structured DAG in order to compute a worst case cost evaluation. The prediction of performance is quite accurate in many cases. Nevertheless, the mandatory use of global synchronizations can lead to inefficient programs.

Skillicorn [16] proposes a more asynchronous version of BSP called MINIBSP. The MINIBSP composition rules allow supersteps to be composed sequentially and in parallel. Therefore, it is a subset-synchronizing language. As BSP, it allows to construct a structured DAG, thereby computing a worst case cost. Nevertheless, partial synchronizations may introduce useless waiting points for some processors.

Culler and Al [2] define a model called LogP, well suited for asynchronous algorithms. The model is based on four parameters which abstractly specify abstractly the target machine. It analyzes the communication scheme in order to deduce the cost associated. In contrast to the $\mathcal{SCL}$-chan cost model, it does not consider the local computation cost, thereby forbidding evaluation of the computation communication overlap. Because of the unstructured communication scheme, it can only captures a specific execution of the program, thereby making difficult the evaluation of "worst case" costs.

8 Conclusion

We presented a complexity model well suited to asynchronous structured message passing. Thanks to our structured communication semantics, we were able to build a complexity function whose definition relies on an induction on the program structure. It yields a symbolic expression encoding the communication scheme. This result is used to evaluate an upper bound of the network load taking into account the asynchronism of communication events. Thanks to this evaluation, we can deduce a symbolic cost associated to a given execution of the algorithm. Note that the complexity can be easily adapted to another modelization of the network by modifying the network load function.

In contrast to synchronous models like BSP, DBSP, CGM [17, 11, 3, 4], it can take into account communication-computation overlapping. Therefore, it can trade off different mapping to efficiently exploit the parallelism of the algorithm. If we do not consider the global network load ($\tau = 0$), our approach models LogP [2]. Note that the second step of the complexity is then useless. But in contrast to LogP which computes a cost for an execution in an unstructured communication scheme, it can cut itself off from a part of the environment in order to compute a worst case cost for a program.

Experimental results on Cray T3E show that cost predictions are comparable to the performances. The complexity model appears to be a good trade-off between usability and effectiveness.

We are currently producing benchmarks on other architectures. The study of algorithms containing while loop structures is under progress. We plan to apply this complexity model to automatic parallelization.

References

[1] BK.R. Apt and E-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[2] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and V. E. Thorsten. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 1–12, May 1993.

[3] P. De La Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. *Lecture Notes in Computer Science*, 1124:352–??, 1996.

[4] F.Dehne, A.Fabri, and A. Rau-Chaplin. Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputer. *ACM 9th Symposium on Computational Geometry*, 1993.

[5] P. Feautrier. Automatic Parallelization in the Polytope Model. In G-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–100. Springer-Verlag, 1996.

[6] I. Foster. *Designing and Building Parallel Programs.* Addison–Wesley, 1994.

[7] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation Series. The MIT Press, 1994.

[8] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Loosely Synchronized Execution Model for a Simple Data-Parallel Language. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96 Parallel Processing*, number 1123 in LNCS, pages 732–741. Springer-Verlag, 1996.

[9] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Structural Clocks for a Loosely Synchronized Data-Parallel Language. In *Second International Conference on Massively Parallel Computing Systems - MPCS'96*, pages 482–489, Ischia, Italy, May 1996. IEEE Computer Society Press.

[10] J. Gabarro J. Balcazar, J. Diaz. *Structural Complexity II.* Springer-Verlag, 1989.

[11] W F McColl. General purpose parallel computing. In A M Gibbons and P Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 93.

[12] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A General but Simple Technique to Handle Asynchronous Data-Parallel Control Structures. In *Fifth Euromicro Workshop on Parallel and Distributed Processing - PDP'97*, pages 189–196, London, United Kingdom, January 1997. IEEE Computer Society Press.

[13] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. $\mathcal{SCL}$-chan: An Asynchronous Data-Parallel Language for Irregular Algorithms. In *Second International Workshop on High-Level Parallel Programming Models and Supportive Environments - HIPS'97 (in conjonction with 11th International Parallel Processing Symposium - IPPS'97)*, pages 64–71, Geneva, Switzerland, April 1997. IEEE Computer Society Press.

[14] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Simple Synchronization and Communication Multi-threaded Library for Automatic Distribution of Irregular Sequential Code. In *Third International Conference on Massively Parallel Computing Systems - MPCS'98*, Colorado Springs, Colorado U.S.A., April 1998. IEEE Computer Society Press.

[15] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Structured Synchronization and Communication Model Fitting Irregular Data Accesses. *Journal of Parallel and Distributed Computing*, 50(1):3–27, 1998.

[16] D. B. Skillicorn. miniBSP: A BSP Language and Transformation System. `http://www.qucis.queensu.ca/home/skill`, 1996.

[17] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.