# Towards Mixed Computation / Communication in Parallel Scientific Libraries

C. Calvin, L. Colombet,
P. Michallon and D. Trystram
LMC-IMAG
46, avenue Félix Viallet
38031 Grenoble Cedex France
*[calvin-colombet-michallon-trystram]@imag.fr*

F. Desprez and B. Tourancheau
Computer Science Department
The University of Tennessee
107 Ayres Hall, Knoxville, TN 37996-1301, USA
*[desprez-btouranc]@cs.utk.edu*

B. Jargot
LIP, CNRS URA 1398
46, allée d'Italie
69346 Lyon Cedex 07 France
*bjargot@lip.ens-lyon.fr*

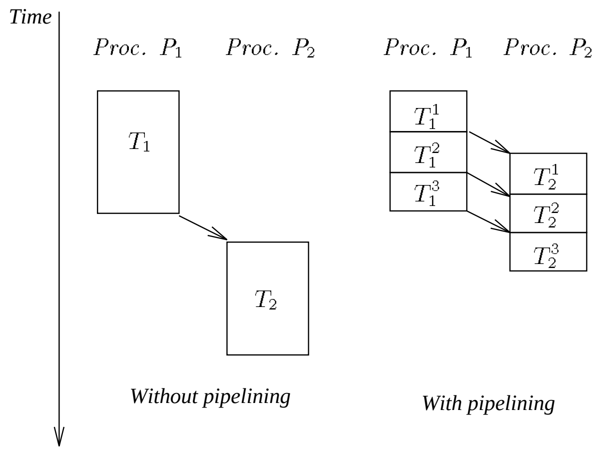
**Abstract**

This paper presents an overlapping technique of communications by computations based on pipelined communications. This allows to improve the execution time of most parallel numerical algorithms. Some simple examples are developed namely, global combine, matrix-vector product and bi-dimensional Fast Fourier Transform, to illustrate the efficiency of this technique. Moreover, we propose an unified formalism to express easily the pipelined versions of these algorithms. Finally, we report some experiments on various parallel machines.
**Keywords**: parallel numerical algorithms, communications.

## 1 Introduction

The large development of parallel scientific computers leads the common users to adapt their programs, to take into account the potential of parallelism of their applications. Since efficient parallel compilers are not available today, the simplest issue, is to use parallel numerical libraries [1]. Given an initial distribution of the data (vectors, matrices, etc.), the computational kernels are identified on each processor. Then, global communication routines are used for data movements. Several such local computation and communication libraries have been developed [1, 9, 10, 19]. However, this way of programming parallel machines is not always efficient because many processors can wait for a long time before receiving data.

The solution that we propose here is based on pipelined data parallel algorithms as defined in [15]. The principle consists in pipelining some computation and communication phases in order to overlap communications by computations. This principle is described on Figure 1. The computational kernels are split into small pieces to transmit faster intermediate results. This allows the processors to receive quickly a part of the data and then, to begin their local computations (in other words, some communications are overlapped by computations. This leads to more parallelism).




Parallel execution of two tasks $T_1$ and $T_2$ on processors $P_1$ and $P_2$.
$T_1$ and $T_2$ are sets of decomposables tasks $T_j^i$, $j = 1, 2$. Each $T_2^i$ depends of $T_1^i$.

Figure 1: Principle of fine grain pipelining

### 1.1 Outline

The goal of this paper is to specify a set of mixed computation/communication routines (namely, broadcast and vector updates, reduction and broadcast, total exchange and dot products, matrix transpose and butterfly computations, etc.) which are useful in most numerical algorithms. The methods are often difficult to express, so we propose a unified formalism that can be used for programming numerical applications. For instance, we propose several well-known numerical kernels that have been developped and rewritten using this formalism. We present in the following some examples, namely, global combine, matrix-vector product and bi-dimensional Fourier transform. For each algorithm, we first analyse the complexity, and compare with the non-overlapped version. After a brief description of the LOCCS library, we give the algorithms of the previous examples rewritten using LOCCS, and give some experiments on parallel machines and workstation networks.

### 1.2 Models and interconnection networks

In the following, we assume that the communication links between processors are bi-directional. We suppose furthermore that a processor can simultaneously send and receive a message on all its links and that during a computation (full-duplex, parallel, all-port model [14, 18]). We also suppose, according to most available parallel computers, that the transmission time of a message of $L$ data between two neighbor processors is: $\beta_c + L\tau_c$, where $\beta_c$ corresponds to a start-up time and $\tau_c$ is the inverse of the bandwidth [14, 17].

We will denote by $P$ the number of processors. The different topologies of the interconnection networks that we are interested in are the bi-directional ring and the hypercube [16].

# 2 Examples

Let us now detail some typical examples.

## 2.1 Global combine on a linear array

The global combine is a collective operation which consists in combining a set of vectors, element by element, on one processor using an associative and commutative operation denoted as $\bigoplus$ (like sum, product, minimum, maximum or boolean operations).
We suppose that each processor contains a vector of $n$ elements. Let us consider a ring of $P$ processors, and denote by $D_{i,j}$ the $j^{th}$ element on the processor $i$. Then a global combine on a root $k$ (which can be any processor) compute : $D_{k,j} = D_{0,j} \bigoplus D_{1,j} \bigoplus \ldots \bigoplus D_{P-1,j}$
For a sake of simplicity, we will consider that $P$ is odd.

A naive algorithm is the following: the farest processors from root $k$ send their vectors to their neighbor which, after combining with their own elements, send the result to their neighbor and so on, until the last partial results reach the root. The total cost of this algorithm is given by :

$$T = \frac{P-1}{2}(\beta_c + n\tau_c) + \frac{P+1}{2}n\tau_a$$

We can improve this algorithm by splitting the vector of each processor into $\mu$ packets of size $\frac{n}{\mu}$. Then, we can apply the same algorithm described above on each packet in order to obtain a maximal use of the communication links and of the processors. Figure 2 shows an example for $P = 7$, $n = 25$ and $\mu = 5$. Notice that, there exist other methods which overlap computations by communications [2]. Although the method described here is not the best possible for this topology, it allows to obtain good performances with a very simple code. The goal here, was only to illustrate the splitting principle.

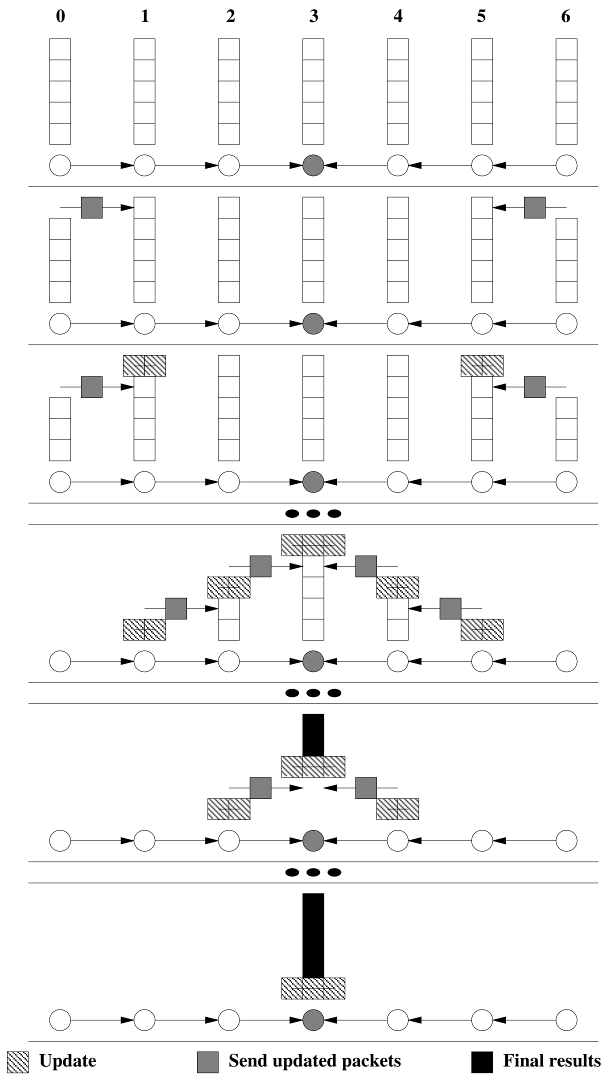



Figure 2: Successive steps of global combine with pipelining on a ring

While using the splitting strategy, the total time of this algorithm is given by :

$$T = \left(\beta_c + \frac{n}{\mu}\tau_c\right) + (P-3)\max\left(\beta_c + \frac{n}{\mu}\tau_c, \frac{n}{\mu}\tau_a\right) + (\mu - 1)\max\left(\beta_c + \frac{n}{\mu}\tau_c, 2\frac{n}{\mu}\tau_a\right) + 2\frac{n}{\mu}\tau_a$$

Where the maximum express the overlap of computation and communication at each step. Obviously, there is an optimal tradeoff for the choice of $\mu$ depending of the machine parameters, and we now analyse the different cases to derive $\mu_{opt}$ (the best splitting as possible).
Several cases occur, depending from the maximum values:

1. If $\beta_c + \frac{n}{\mu}\tau_c \leq \frac{n}{\mu}\tau_a$. Then the total time of the algorithm is equal to:
$$T = \beta_c + \frac{n}{\mu}\tau_c + (P - 3 + 2\mu)\frac{n}{\mu}\tau_a \Longrightarrow \mu_{opt} = \frac{n(\tau_a - \tau_c)}{\beta_c}$$

2. If $\frac{n}{\mu}\tau_a < \beta_c + \frac{n}{\mu}\tau_c \leq 2\frac{n}{\mu}\tau_a$. Then the total time of the algorithm is equal to:

$$T = (P-2)\left(\beta_c + \frac{n}{\mu}\tau_c\right) + 2n\tau_a \Longrightarrow \mu_{opt} = \frac{n(2\tau_a - \tau_c)}{\beta_c}$$

3. If $\beta_c + \frac{n}{\mu}\tau_c > 2\frac{n}{\mu}\tau_a$. Then the total time of the algorithm is equal to:

$$T = (P-3+\mu)\left(\beta_c + \frac{n}{\mu}\tau_c\right) + \frac{2n}{\mu}\tau_a \Longrightarrow \mu_{opt} = \sqrt{\frac{n((P-3)\tau_c + 2\tau_a)}{\beta_c}}$$

For most distributed-memory parallel machines, the third case is the more realistic. Then, the total execution time is given by:

$$T = (P-3)\beta_c + n\tau_c + 2\sqrt{\beta_c((P-3)\tau_c + 2\tau_a)n}$$

The time is asymptotically $\mathcal{O}(n\tau_c)$.

## 2.2 Matrix-vector product on a ring

The matrix-vector product is one of the most important operations in iterative methods of linear algebra [11]. We consider the computation of $v = Ax$ where $A$ is a $n$ by $n$ matrix and $x$ and $v$ are two vectors of dimension $n$. Each component of vector $v$ is computed as :

$$v[i] = \sum_{j=1}^{n} A[i,j]\ x[j] \tag{1}$$

We assume a balanced allocation of matrix $A$ formed by $P$ blocks of $\lceil \frac{n}{P} \rceil$ or $\lfloor \frac{n}{P} \rfloor$ rows. Similarly, each processor owns $\lceil \frac{n}{P} \rceil$ or $\lfloor \frac{n}{P} \rfloor$ components of the vector $x$. (Remark that a processor can equivalently store consecutive rows, wrap rows or cyclic rows [12]). To simplify the presentation, we will consider that P is odd and $\frac{n}{P}$ is integer.

As shown in equation (1), computing each component of vector $v$ requires the knowledge of the whole vector $x$. So, each processor must send its $\frac{n}{P}$ components of vector $x$ to all the other processors. The data movement corresponds to a global communication called gossiping or ATA (for All To All) [17]. This matrix-vector product algorithm can be described by the following code:

```
begin
   ATA of size n/P (part of vector x);
   do in parallel, on each processor
      computation of n/P dots of size n
end
```

Figure 3: Usual parallel algorithm of matrix-vector product.

The ATA with messages of size $\frac{n}{P}$ costs $\frac{P-1}{2}(\beta_c + \frac{n}{P}\tau_c)$ units of time [17] on a bi-directional ring. The computation of $\frac{n}{P}$ dots of size $n$ costs $\frac{n^2}{P}\tau_a$ units of time. Then, the standard parallel matrix-vector product implemented on a ring has a complexity of:

$$T = \frac{n^2}{P}\tau_a + \frac{P-1}{2}\left(\beta_c + \frac{n}{P}\tau_c\right) \tag{2}$$

An interesting solution to approach the minimum execution time is to overlap the communications with computations. During the ATA, the processors loose some time. Indeed, they could have started the computation of their initial $\frac{n}{P}$ data, during the ATA. An idea to improve the previous solution is to decompose the global ATA in several local (neighbor to neighbor) communications in order to decrease the communication overhead. A processor can receive new data during the computation of previously received data.

More precisely, at the beginning, each processor owns $\frac{n}{P}$ rows of matrix $A$ and $\frac{n}{P}$ components of vector $x$. It can compute $\frac{n}{P}$ partial dot products $v_i$ with its $\frac{n}{P}$ components of $x$. Thus, it can execute $(\frac{n}{P})^2$ floating-point and receive $\frac{n}{P}$ data from each of its direct neighbor. All the next steps consist in the computation of $2\frac{n}{P}$ dots of size $\frac{n}{P}$ and a communication of $2\frac{n}{P}$ elements of vector $x$ (see figure 4).

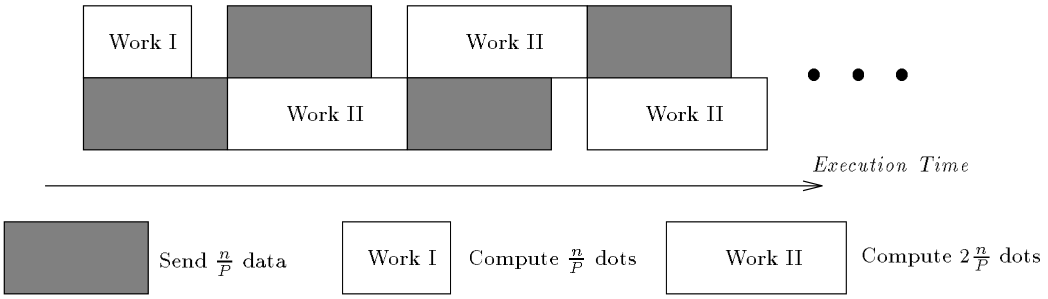



Figure 4: Execution chart of one processor

As for previous example, three cases occur depending on the communication cost of $\frac{n}{P}$ data. The simple one is obtained when $\frac{n^2}{P^2}\tau_a \geq \beta_c + \frac{n}{P}\tau_c$ and all the communications are overlapped. Figure 4 represents the intermediate case when $\frac{n^2}{P^2}\tau_a \leq \beta_c + \frac{n}{P}\tau_c \leq 2\frac{n^2}{P^2}\tau_a$. Let us remark that in the worst case corresponding to "large" communications ($2\frac{n^2}{P^2}\tau_a \leq \beta_c + \frac{n}{P}\tau_c$), this strategy can be improved [6]. The idea is to send the maximum data as possible that is $\mu_1 = \left\lfloor \frac{n^2}{P^2}\frac{\tau_a}{\tau_c} - \frac{\beta_c}{\tau_c} \right\rfloor$. This process defines a strictly decreasing series $\mu_t = \left\lfloor \frac{n}{P}\frac{\tau_a}{\tau_c}2\mu_{t-1} - \frac{\beta_c}{\tau_c} \right\rfloor$ (see [6] for more details).

The final execution time of a matrix-vector product with communication overlap is:

$$T = max\left[2\frac{n^2}{P^2}\tau_a + \frac{P-1}{2}\left(\beta_c + \frac{n}{P}\tau_c\right)\ ,\ (P-1)\frac{n^2}{P^2}\tau_a + max\left(\frac{n^2}{P^2}\tau_a, (\beta_c + \frac{n}{P}\tau_c)\right)\right].$$

## 2.3 Bi-dimensional FFT

There exist several methods for computing in parallel the bi-dimensional Fast Fourier Transform (FFT_2D) depending on the data allocation. We study here the Transpose Split (TS) method which uses a row distribution [5]. Let us first recall briefly the problem of computing the bi-dimensional

Discrete Fourier Transform [3, 5]:

$$X(j_1, j_2) = \sum_{k_1=0}^{n-1} \left( \sum_{k_2=0}^{n-1} x(k_1, k_2) \times \exp\left( \frac{-2i\pi(k_1.j_1 + k_2.j_2)}{n} \right) \right) \qquad j_1, j_2 = 0 \ldots n-1$$

This equation can be transformed in order to come down to the computation of two classical mono-dimensional FFT (FFT_1D) [3, 5].

The Transpose Split algorithm is the following: initially each processor has a set of rows of the original matrix. After having computed the FFT_1D on each row, a transposition is performed on the matrix. Then, another FFT_1D is done on the resulting matrix (see Figure 5(a) for the general algorithm). In this algorithm, the overhead due to the matrix transpose leads to poor efficiencies. Our aim is to pipeline the communication phases, i.e. transpose a row a soon as it has been computed [4].

Each processor has to compute $\frac{n}{P}$ FFT and then distribute these data to the other processors. Let us split these $\frac{n}{P}$ rows into packets of size $\mu$. Thus we can overlap the communication of packet $i-1$ by the computation of packet $i$. The algorithm is described in Figure 5(b).

```
begin
   Compute FFT_1D on the rows
   Transpose the matrix
   Compute FFT_1D on resulting rows
end
```

(a) Algorithm without pipelining

```
begin
   Compute FFT_1D on the first packet
   for i = 1 to n/μ − 1
      Transpose packet i − 1
      Compute FFT_1D on packet i
   endfor
   Transpose packet n/μ − 1
   Compute FFT_1D on resulting rows
end
```

(b) Algorithm using pipelining

Figure 5: Bi-dimensional FFT algorithms without and with pipelining.

We present in Figure 6 the execution chart of one processor.

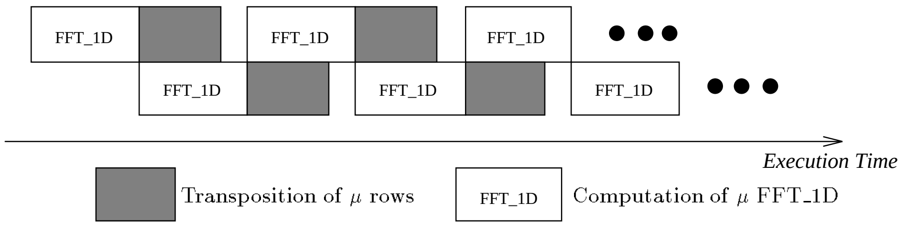



Figure 6: Execution chart of one processor

It is well-known that the cost of the FFT_1D on a row of $n$ elements, denoted by $T_{FFT}(n)$, is equal to $n\log_2(n)\tau_{fft}$ ( where $\tau_{fft}$ is the cost of one basic arithmetic butterfly operation in the

FFT algorithm). We assume that the cost of the internal transposition is negligible in regard to the computation of the FFT_1D. For a row-wise distribution of the matrix, the communication scheme which corresponds to a matrix transposition is a personalized-all-to-all communication (PATA), also called multi-distribution [13, 17]. We denote by $T_{com}(n)$ the communication cost of vector of size $n$.

We now detail the communication cost on a ring: $T_{com}(n) = \lfloor \frac{P}{2} \rfloor \beta_c + \frac{nP}{8}\tau_c$ ([17]). As in the previous section, we can determine, the minimum size of the packet, which allows the best overlap of communications by computations.

$$\mu T_{FFT}(n) > T_{com}(\mu n) \Rightarrow \mu > \frac{\lfloor \frac{P}{2} \rfloor \beta_c}{n\left(\log_2(n) - \lfloor \frac{P}{8} \rfloor\right)}$$

This leads to the total complexity:

- With no overlap $\quad T = 2\frac{n^2}{P}\log_2(n)\tau_{fft} + \lfloor \frac{P}{2} \rfloor \beta_c + \left\lfloor \frac{n^2}{8} \right\rfloor \tau_c$ [13, 17]
- With total overlap $\quad T = 2\frac{n^2}{P}\log_2(n)\tau_{fft} + \lfloor \frac{P}{2} \rfloor \beta_c + \left\lfloor \frac{n\mu P}{8} \right\rfloor \tau_c$

The same kind of analysis can be made on a hypercube. It leads to the following results:

- With no overlap $\quad T = 2\frac{n^2}{P}\log_2(n)\tau_{fft} + \log_2(P)\beta_c + \left\lfloor \frac{n^2}{2P} \right\rfloor \tau_c$ [14]
- With total overlap $\quad T = 2\frac{n^2}{P}\log_2(n)\tau_{fft} + \log_2(P)\beta_c + \lfloor \frac{\mu n}{2} \rfloor \tau_c$

Notice that, even if computations are not completely overlapped by computations, the overlapped algorithm is still more efficient than any non-overlapped algorithm [4].

All the presented algorithms using pipelining are not very easy to implement. Now, we present briefly the LOCCS library which allows to express them in a simpler way.

# 3 Brief description of the LOCCS library

In this section we specify the different routines available in this library and we detail one of them: OTA, the well-known broadcast communication. We also present experimental results on the Intel iPSC/860 and Paragon machines.

## 3.1 Presentation of the different routines

The LOCCS (Low Overhead Communication and Computation Subroutines) is a general library for coarse grain pipelining on different parallel machines [8]. There are a total of 8 routines, corresponding to the classical communication patterns, that can be associated to computation procedure, in order to be overlapped and thus to decrease the communication overhead.

| Span | Routine | Communication pattern |
|---|---|---|
| Point-to-point | `loccs_oto` | One-to-one |
| | `loccs_exchange` | Exchange |
| | `loccs_shift` | Shift |
| Global | `loccs_ota` | Broadcast |
| | `loccs_pota` | Scattering |
| | `loccs_ato` | Gathering |
| | `loccs_ata` | All-to-all |
| | `loccs_pata` | Multi-scattering |

Table 1: LOCCS routines and communication patterns.

For each routine, the communication phase is a call to a non-blocking high-level communication routine on a packet of size $\mu$. This ensures the portability of the routine. The programmer, who gives the code of the computation tasks, does not have to know the underlying algorithm used for the communications.

### 3.2 One LOCCS routine: `loccs_ota`

In this section, we give an example of the broadcast-overlap routine: `loccs_ota`. The processors involved can have two tasks to do, that we specified in the calling sequence, one before the communication and one after. A call to the `loccs_ota` routine is realized by:

`loccs_ota(me, ota, buffer,` $\mu$`, size,` $job_b$`,` $job_a$`)`

We have different structures to define environment and buffers.

| | |
|---|---|
| `me` : | Logical number of the caller `loccs_ota`, |
| `ota` : | A structure giving the index of the root, |
| `buffer` : | A structure for message buffers, |
| $\mu$ : | Packet size, |
| `size` : | Total size of the message, |
| $job_b$, $job_a$ : | Tasks which will be executed before and after a communication step. |

### 3.3 Implementation

One interest of this library is its portability, and thus the portability of efficient codes. We have chosen to implement one subset of the routines on various machines and environments. The first version of the library is written in C but we are working on two other versions in C++ and Fortran. Table 2 gives the name of the routines implemented on different machines.

| Architecture | Available routines | Environment used |
|---|---|---|
| Intel iPSC/860 | `loccs_oto, loccs_ota,` `loccs_exchange, loccs_shift` | NX/PICL |
| Intel Paragon | `loccs_oto, loccs_ota` `loccs_exchange, loccs_shift` | NX |
| Workstation network | `loccs_oto` / | PVM 3.2 Trollius LAM |

Table 2: The different implementations of LOCCS routines.

Several experiments were done on the Intel iPSC/860 and Paragon machines. The curves obtained give an idea of the execution time of the routines and show the optimal size of the packets. The tested routines are `loccs_ota` and `loccs_oto`. The tasks use loops of computation operations and we make their costs vary to have an idea of the impact of the machines and algorithm parameters on the execution time.

Figure 7 shows different executions of the `loccs_ota` routine on the iPSC/860. We modify the ratio between the two tasks (before and after communication) for the `loccs_ota` on the iPSC/860, in order to know its influence. The total cost of computation is 30 units where one unit is an addition on each element of the current packet. We have executed the routine for the ratios $job_b/job_a$ equal to 0.5, 1 and 2. For each ratio, the packets size varies, from one element to the total size of the message. The "hills" on the curves are due to the size of the last packet. We can see that the best execution time is obtained with tasks of equal sizes.

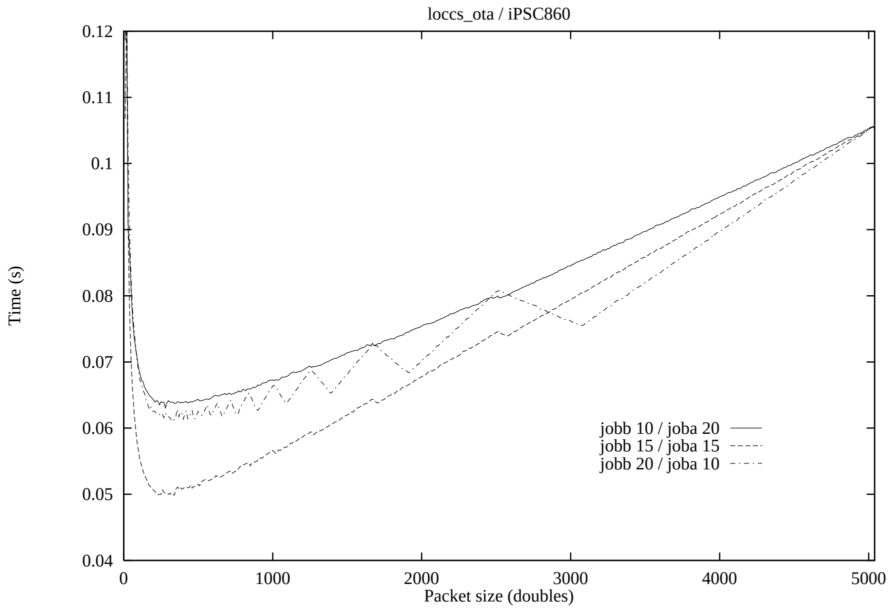



Figure 7: Execution of `loccs_ota` on iPSC/860 using 4 processors.

Figure 8 presents the same kind of experiments for the `loccs_oto` routine on the iPSC/860 and Paragon machines.

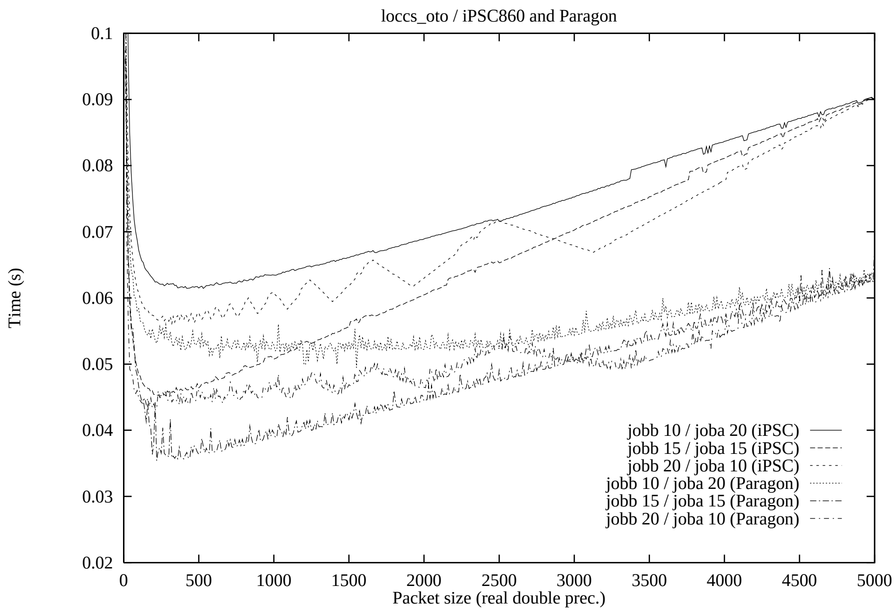



Figure 8: Some executions of `loccs_oto` on Intel iPSC/860 and Paragon machines.

# 4 Application of LOCCS to the previous examples

## 4.1 Global combine using LOCCS

The global combine operation can be easily rewritten using calls to a LOCCS routine. This routine, called `loccs_shift`, performs shifts on a processor between two sets of processors. These two sets contain the logical numbers of processors which send or receive data from the processor calling the routine.

In Figure 9, we give the algorithm for one processor of the ring. The initialization of the parameters of the LOCCS routine are done in the `init_tabs`, `init_jobs` and `initShift` routines. The complete program has been given in [7]. The `closeShift` routine releases the buffers used by the LOCCS routines.

```
begin
⋮
   /* initialization of the LOCCS routine parameters */
init_tabs(shift.senders, shift.receivers)
init_jobs(job_b,job_a)

err = initShift(&buffer, μ, sizeof(double), type, me, &shift)
loccs_shift(me, &shift, &buffer, μ, taille, job_b, job_a)
closeShift(&buffer,type)
⋮
end
```

Figure 9: Global combine operation on a ring using `loccs_shift`

The global combine operation with pipelining has been implemented using the loccs_shift routine on an iPSC/860 with different sizes of packets. Figure 10 represents an execution of the global combine on matrices of size 100 on seven processors. The three space-time graphs correspond to the execution with 1, 4 and 10 packets. The decrease of the total time with increase in the number of packets is obvious.

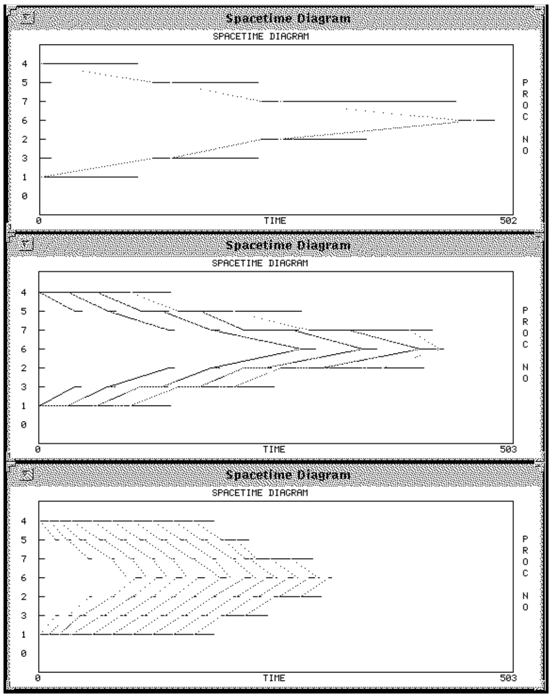



Figure 10: Execution of the global combine operation using `loccs_shift` on an iPSC/860.

### 4.2 Matrix-vector product using LOCCS

The LOCCS version of this algorithm is given on figure 11. Note that the job before WorkI is used only once to initialize the pipeline. The packet size equals to $\frac{n}{P}$, corresponds to a " happy compromise " of overlapping. A detailed study of the optimal size of packets is given in [6].

```
begin
   loccs_ata(me, ata, buffer, n/P, n, WorkI, WorkII)
end
```

Figure 11: Parallel matrix-vector product using LOCCS

### 4.3 Bi-dimensional Fourier Transform

The algorithm that we obtain with a simple use of the loccs_pata is given on Figure 12. For the first phase, we overlap the transposition of the matrix by the computation of the mono-dimensional FFT of the remaining rows and the internal transposition of the received packets (job int_transp). Notice that we can use a different algorithm for the FFT_1D on the resulting columns, and suppress the job int_transp when the matrix is completely transposed.

The computation of the FFT_1D on the resulting rows is done after the call to the loccs_pata.

Notice that the parameter $\mu$ must be a multiple of the size of a row, because of the dependences in the computation of the FFT_1D.

```
begin
   μ = rn
   loccs_pata(me, pata, buffer,μ, FFT1D,NOJOB)
   FFT1D(i,n,n/P)
end
```

Figure 12: Transpose split algorithm using `loccs_pata` routine.

We present on figure 13 some experiments of the both methods (with no overlapping and with overlapping) on 8 IBM RS6000 using PVM 3.2.6. Remark that the gain of time is about 37.5% better for the overlapped method.
The choice of the packet size has been determined experimentally using an adaptive method.

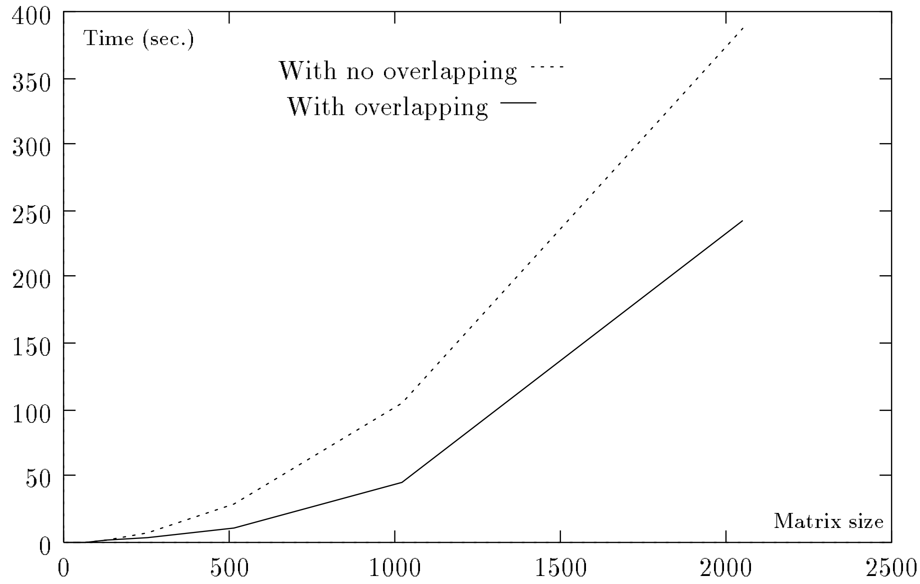



Figure 13: Comparison of the overlapped and not overlapped method of the FFT2d on 8 workstations for various sizes of matrices.

# 5 Conclusion

This paper have dealt with the improvement of basic numerical kernels by fine grain pipelining techniques which allows the overlapping of communications. The library that we have developed (LOCCS) leads to a simple formalism for expressing pipelined data parallel algorithms.
We have presented in this paper three simple examples which use this formalism. It can be applied to many other fundamental numerical methods like Gaussian elimination, conjugate gradient, singular value decomposition, etc. Some routines have been implemented using C language on some parallel

machines (iPSC, Paragon, workstation networks), and would be extended to other machine like IBM SP1, using several interface languages (Fortran and C++).

## References


[1] E. Anderson and al. *LAPACK Users's Guide.* SIAM, 1992.

[2] M. Barnett, R. Littlefield, D.G. Payne, and R. Van De Geijn. On the Efficiency of Global Combine for 2-D Mesh With Wormhole Routing. Technical report, University of Texas, Austin, 1993.

[3] A. Brass and G.S. Pawley. Two and three dimensional FFTs on highly parallel computers. *Parallel Computing*, 3:167–184, 1986.

[4] C. Calvin and F. Desprez. Minimizing Communication Overhead Using Pipelining for Multi-Dimensional FFT on Distributed Memory Machines. In Evans Joubert Liddell, editor, *Proceedings of the International Conference on Parallel Computing '93*, Advances in Parallel Computing. North Holland, 1993.

[5] C. Y. Chu. Comparison of Two-Dimensional FFT Methods on the Hypercube. In Geoffrey and Fox, editors, *The Third Conference on Hypercube Concurrent Computers and Applications*, volume 2, 1988.

[6] L. Colombet, P. Michallon, and D. Trystram. Matrix Vector Product on the Ring Topology with Overlapping Communications. Technical report, LMC- IMAG, Grenoble, January 1994.

[7] F. Desprez. *Procédures de Base pour le Calcul Scientifique sur Machine Parallèle à Meémoire Distribuée.* PhD thesis, Institut National Polytechnique de Grenoble, January 1994. LIP ENS-Lyon.

[8] F. Desprez and B. Tourancheau. LOCCS: Low Overhead Communication and Computation Subroutines. In *High Performance Computing and Networking Conference - Amsterdam.* Elsevier, May 1993.

[9] J.J. Dongarra, R.A. Van De Geijn, and R.C. Whaley. Two Dimensional Basic Linear Algebra Communication Subprograms. In Dongarra and Tourancheau, editors, *Environments and Tools For Parallel Scientific Computing - St Hilaire du Touvet*, volume 6 of *Advances in Parallel Computing.* North Holland, September 1992.

[10] G.A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley. PICL: A Portable Instrumented Communication Library. Technical Report ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[11] G.H. Golub and C.F. Van Loan. *Matrix Computation.* The John Hopkins University Press, 1989. Second edition.

[12] S.L. Johnsson. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *Journal of Parallel and Distributed Computing*, pages 133–172, 1987.

[13] S.L. Johnsson and C.T. Ho. Algorithms for Matrix Transposition on Boolean n-cube Configured Ensemble Architectures. *SIAM Journal Matrix Anal. Appl.*, 9(3):419–454, July 1988.

[14] S.L. Johnsson and C.T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transaction on Computers*, 38(9):1249–1268, 1989.

[15] C.T. King, W.H. Chu, and L.M. Ni. Pipelined Data Parallel Algorithms - Concept and Modeling. In *International Conference on Supercomputi ng*, pages 385–395, July 1988.

[16] F. T. Leighton. *Introduction to Parallel Algorithms and Parallel Architectures: Arrays - Trees - Hypercubes.* Morgan Kaufman Publishers, Inc., 1992.

[17] Y. Saad and M. Schultz. Data Communication in Parallel Architectures. *Parallel Computing*, (11):131–150, 1989.

[18] Q. F. Stout and B. Wagar. Intensive Hypercube Communication, Preearranged Communication in Link-Bound Machines. *Journal of Parallel and Distributed Computing*, Vol. 10:167–181, 1990.

[19] D.W. Walker. Standards for Message-Passing in a Distributed Memory Environment. Technical Report ORNL/TM-12147, Oak Ridge National Laboratory, August 1992.