LOCCS: Low Overhead Communication and Computation Subroutines*

F. Desprez and B. Tourancheau
LIP-IMAG
Unité de Recherche Associée 1398 du CNRS
Ecole Normale Supérieure de Lyon
46, Allée d'Italie
69364 Lyon Cedex 07
France
e-mail: [desprez,btouranc]@lip.ens-lyon.fr

TR-92.44

December 18, 1992

Abstract

Our aim is to provide one set of efficient basic subroutines for scientific computing which include both communications and computations. The overlap of communications and computations is done using asynchronous pipelining to minimize the overhead due to communications. With this set of routines, we provide to the user of parallel machines an easy SPMD type and efficient way of programming. The main purpose of theses routines is to be used in linear algebra applications but also in other fields like image processing or neural networks.

*This work was partially supported by ARCHIPEL S.A. under contract 820542, by the CNRS and the DRET

1 Introduction

Libraries of routines have been proven to be the only way for efficient and secure programming. In scientific parallel computing, the most commonly used libraries are the BLAS, BLACS, PICL and the one provided by vendors. These building blocks allow the portability of codes and an efficient implementation on different machines.

The development of sets of subroutines is now one of the challenge of the construction of software tools for parallel distributed memory machines.

Every computer manufacturer includes in its software package sets of computation and sets of communication subroutines. Hence, parallel programs developers do not have to hand-code the low level routines anymore.

But communication and computation routines are still distinct routines and often the overlap of communications by computations is not possible, because of a synchronous mode of execution. This lack of overlap leads to an unacceptable overhead of communications, leading to poor efficiencies. Thus, the high level routines are not always used by programmers who have to hand-code pipeline methods to obtain reasonably efficient applications. To improve this situation, one solution is to have non-blocking communications routines. But in this case, communication and computation routines have to be totally independent. A second solution is to mix communications and computations, using pipelining in a given pre-programmed routine to obtain the lowest possible overhead for the communication.

The goal of this paper is to specify such a set of routines corresponding to the well known local and global communication procedures associated to a given non defined computation task. We call this library the LOCCS library (Low Overhead Communication and Computation Subroutines).

In the first two sections, we present some of the basic sets of communications and computations subroutines, available for the Intel iPSC/860 hypercube. In the third section, we present our set of routines. In the last section and before some concluding remarks, we present some applications of our subroutines.

2 Communications libraries

In this section, we present the most used communication and linear algebra libraries available on the Intel iPSC/860 hypercube.

2.1 Intel communication library

Intel provides on the iPSC/860 a library for the basic communications operations. It includes neighbor synchronous and asynchronous communications, broadcast and global combine operations.

2.2 PICL library

The PICL library [15] (Portable Instrumented Communication Library) has been developed at Oak Ridge National Laboratory and provides the same routines as the Intel library and routines for the set-up of the machine.

One interesting feature of this library is that it supports the generation of traces for ParaGraph, the parallel programs visualization tool [17].

This library is based on three sets of routines: one set of low levels communication routines, one set of high level global communication routines and a set of routines for the additional trace generation.

The high level routines, which are built on top of the low level routines, include synchronization, broadcast and topology embedding (to define the underlying topology mapped on the hypercube).

2.3 BLACS library

The BLACS [1, 11, 12] (Basic Linear Algebra Communication Subroutines) are dedicated to communication operations involved in the parallelization of the level 3 BLAS and LAPACK libraries.

As PICL, the BLACS include low level routines and high levels routines. The low level routines are SD (send a message), RV (receive a message), whereas the high level routines are BS (broadcast a message) and BR (receive a broadcasted message).

The BLACS include global operators too, such as maximum (MAX), minimum (MIN) and summation (SUM).

The previous routines act on different data structures and different data types. The structures are essentially rectangular and trapezoidal matrices. The types are integer, real, double precision, complex and double complex.

3 Linear Algebra libraries

In this section, we present some of the linear algebra libraries available for the Intel iPSC/860 hypercube. These libraries are divided in two sets: the sequential ones and the parallel ones.

3.1 Sequential libraries

For the i860, which is the CPU of the Intel hypercube, the well known BLAS [10] (Basic Linear Algebra Subroutines) are available from different vendors. These routines include scalar-vector operations (level 1 BLAS), vector-matrix operations (level 2 BLAS) and matrix-matrix operations (level 3 BLAS).

On the i860, they are assembly coded to get the best of the superscalar architecture and reduce the data movements in the memory.

The performances in Mflops obtained with such routines are far from the peak performances of the i860 (approximately the half) but also far from C coded routines (approximately 8 times faster). The available compilers are not able to manage the data access for this processor.

The LAPACK library [2, 3, 4] (Linear Algebra PACKage), which makes an intensive use of the BLAS, can be compiled for the i860 and thus leads to interesting performances for most of the basic linear algebra problems.

3.2 Parallel libraries

In Oak Ridge Laboratory and the University of Tennessee, intensive researches have been started to change sequential BLAS and LAPACK into distributed memory routines [7, 8]. The parallel version of LAPACK will be called ScaLAPACK (Scalable LAPACK). The growing interest of scientists for the distributed memory parallel machines like iPSC/860, nCube, CM5, KSR or Paragon leads to an obvious need of such routines to solve efficiently, and with "simple" codes their number crunching problems.

4 Low Overhead Communication and Computation Subroutines

4.1 Introduction

Our aim is to provide a set of routines which use pipelining and overlapping to minimize the communication overhead. In parallel linear algebra algorithms, performances are often obtained by mixing non-blocking communication routines with numerical routines. Today, most of the high level communication routines are blocking to be secure but this leads to a important overhead.

By mixing computations and communications, we ensure that the overhead will be minimized, even if the computation cost is less than the communication cost (like for the transposition algorithm). Of course, these routines are more efficient. The programmer using these routines has to know well the dependences between communication and computation parts within his code to obtain accurate results and good performances. All the routines are blocking until completion allowing a SPMD[1]-like programming style.

We assume that computations can overlap communications. Thus the communications libraries must provide a way to have this overlap. On the Intel iPSC/860, this is done via non-blocking send and receive. A test routine to know the end of a communication must also be provided (for example isend, irecv and wait on the Intel).

Another important assumption is that a message can be sent to any non-adjacent node with a negligible overhead. This is true on the iPSC/860 target machine [13].

We suppose that the communication involved can be of high level (such as one-to-all, all-to-all, ...). The LOCCS routines are asynchronous.

4.2 The Macro pipeline

Usually, the parallel algorithms use patterns which correspond to figure 1 (a stands for *after* communication, b for *before* communication and n for *no* data dependence). Our aim is to provide routines realizing an overlap of the communications by the computations for the JOB_{bd}, Com and JOB_{ad} parts.

JOB_{bnd}	A computation part with no dependences with the communication Com
JOB_{bd}	A computation part with dependences with the communication Com
Com	A communication part
JOB_{ad}	A computation part with dependences with the communication Com
JOB_{and}	A computation part with no dependences with the communication Com

Figure 1: Classical scheme of a parallel algorithm

In the following, for sake of simplicity, we will call Job_{bd}, job_{before} and Job_{ad} job_{after}.

The pipeline or macro pipeline method was described in [9] for the rank-2k update algorithm and in [19] for the Gaussian elimination. The method consists in breaking the message to be communicated into packets of size ν and sending the packets one after another while computing. The macro-pipeline splits the computation and the communication procedures into smaller parts. As we realize an overlap of communications by computations, we reduce the overhead due to communications.

[1]Single Program Multiple Data

One given parameter is ν which the size of a packet used in the pipeline. This size has to be computed before the call to the routine. Its value must be computed in order to have the best overlap between communication and computation.

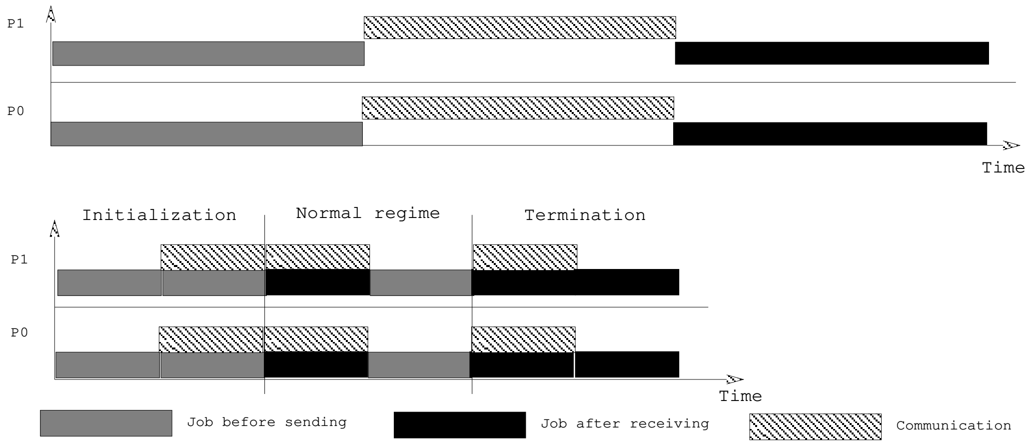

Figure 2: Synchronous and macro_pipeline execution times on two nodes

Figure 2 describes the macro-pipeline method between two nodes. The gain obtained with the macro-pipeline is obvious compared to the classical synchronous scheme.

4.3 Algorithm description

The algorithm is given on figure 3. An overlap of communication by computation is not possible because of the dependences between the jobs and the communication leading to a synchronous execution. To minimize the communication overhead we use macro-pipelining where the message to be sent is split in parts of size ν giving the algorithm of figure 4.

JOB_{before}(A, size_of_A)
com(A, size_of_A)
JOB_{after}(A, size_of_A)

Figure 3: Restricted classical algorithm

Notice that this algorithm is hard to program. With the routines described in the following, the user will just give the size of the packets and a packet oriented version of job_{before} and job_{after} as parameters.

4.4 Parameters

Before the description, we give more informations about the parameters which will be used.

- **topology**: The underlying topology can be given if it is known in order to optimize the communication algorithm. The topologies that will be available are grid, torus, ring or

job_{before} (0, ν, subblock of A)
in // do
  com(subblock 0 of A, ν)
  job_{before} (1, ν, subblock of A)
enddo
	Initialization

for i=1 to number_of_packets-1
  in // do
    job_{after} (i-1, ν, subblock of A)
    com(subblock i of A, ν)
    job_{before} (i+1, ν, subblock of A)
  enddo
endfor
	Normal regime

in // do
  com(subblock number_of_packets-1 of A, ν)
  job_{after} (number_of_packets-2, ν, subblock of A)
enddo
job_{after} (number_of_packets-1, ν, subblock of A)
	Terminaison

Figure 4: Macro pipeline algorithm

hypercube. This correspond to most of the parallel distributed memory machines on the market.

- ν: is the size of a packet of the macro-pipeline. It has to be computed before the call to the LOCCS routine and should lead to the best overlap of communications by computations. Its value depends of the complexity of the computation and communication parts, the performance parameters of the target machine.

- **The jobs**: are the computation parts of the routine.

  They are divided in two distinct parts: job_{before} and job_{after}.

  Job_{before} is the computation which should be done by a node before the sending of the message.

  Job_{after} is the computation which should be done by receivers after having received a packet.

  At a given time, a job acts on the i^{th} packet of size ν. This packet is stored in a buffer. The user must declare these three parameters at the first position of the packet oriented version of JOB that will be passed as a parameter of the LOCCS routine. The LOCCS routine will then update the packet number during its execution (see figure 4). Notice that the reception buffer should be passed to a job_{after} and the emission buffer should be passed to a job_{before}.

- **NOJOB**: For example, if there is no job_{before}, **NOJOB** will replace job_b and its following parameters in the call.

4.5 One-to-one subroutine

Given two nodes (adjacents or not), the loccs_oto sends a message from a node to an other one, after the computation of a job_{before}. Another computation has to be done after the reception. A call to the **loccs_oto** routine is realized by:

- loccs_oto(topo, sdr, rcv, ν, $buffer$, size, job_b, nbpb, ..., job_a, nbpa, ...)

where:

topo	the underlying topology
sdr	logical number of the sender
rcv	the logical number of the receiver
ν	the size of a packet
$buffer$	messages buffers
size	total size of the message
job_b	computations to be done on the message before the communication
nbpb, ...	number of parameters of job_b followed by the parameters (excepted i, ν and buffer)
job_a	computations to be done on the message after the communication
nbpa, ...	number of parameters of job_a followed by the parameters (excepted i, ν and buffer)

Notice that $buffer$ is composed of two buffers of size ν contigous in memory to be able to work with one buffer while communicating with the other one.

4.6 Exchange subroutine

This routine is like the loccs_oto routine but the communication is full-duplex, we have an exchange of two messages. A call to the **loccs_exchange** routine is realized by:

- loccs_exchange(topo, n1, n2, ν, buf_{em}, buf_{rec}, size, job_b, nbpb, ..., job_a, nbpa, ...)

where:

topo	the underlying topology
n1	id of the node calling the routine
n2	id of the other node involved
ν	the size of a packet
buf_{em}	emission buffers
buf_{rec}	reception buffers
size	total size of the message
job_b	computations to be done on the message before the communication
nbpb, ...	number of parameters of job_b followed by the parameters (excepted i, ν and buf_{em})
job_a	computations to be done on the message after the communication
nbpa, ...	number of parameters of job_a followed by the parameters (excepted i, ν and buf_{rec})

4.7 Broadcast subroutine

The broadcast[2] operation is involved in a lot of algorithms, especially in linear algebra, for example in all the factorizations like LU, Cholesky and their derivatives. A call to the **loccs_ota** routine is realized by:

[2] one-to-all

- loccs_ota(topo, root, ν, buf_{em}, buf_{rec}, size, job_b, nbpb, ..., job_a, nbpa, ...)

where:

root	root of the broadcast
topo	underlying topology
ν	the size of a packet
buf_{em}	emission buffer
buf_{rec}	reception buffer
size	total size of the message
job_b	computations to be done on the message before the communication
nbpb, ...	number of parameters of job_b followed by the parameters (excepted i, ν and buf_{em})
job_a	computations to be done on the message after the communication
nbpa, ...	number of parameters of job_a followed by the parameters (excepted i, ν and buf_{rec})

As the root can have the same behaviour as the other nodes, it receives a copy of each packet of the broadcasted message in its buf_{rec}. The other nodes do not need a buf_{em}.

4.8 Scattering subroutine

The scattering[3] operation is involved in a lot of algorithms to distribute data on the processors. We assume that the total size of the message is a multiple of the number of nodes. A call to the **loccs_pota** routine is realized by:

- loccs_pota(topo, root, ν, buf_{em}, buf_{rec}, size, job_b, nbpb, ..., job_a, nbpa, ...)

where:

root	root of the scatter
topology	underlying topology
ν	the size of a packet
buf_{em}	emission buffer
buf_{rec}	reception buffer
size	total size of the message
job_b	computations to be done on the message before the communication
nbpb, ...	number of parameters of job_b followed by the parameters (excepted i, ν and buf_{em})
job_a	computations to be done on the message after the communication
nbpa, ...	number of parameters of job_a followed by the parameters (excepted i, ν and buf_{rec})

4.9 Gathering subroutine

The gathering[4] operation is involved in a lot of algorithms by example to gather results of a computation from all the nodes on the root. Global combine routines like global-sum can be written using the routine **loccs_ato**.

A call to the **loccs_ato** routine is realized by:

[3] personalized one-to-all
[4] all-to-one

- loccs_ato(topo, root, ν, buf_{em}, buf_{rec}, size, job_b, nbpb, ..., job_a, nbpa, ...)

where:

root	root of the gather
topology	underlying topology
ν	the size of a packet
buf_{em}	emission buffer
buf_{rec}	reception buffer
size	total size of the message
job_b	computations to be done on the message before the communication
nbpb, ...	number of parameters of job_b followed by the parameters (excepted i, ν and buf_{em})
job_a	computations to be done on the message after the communication
nbpa, ...	number of parameters of job_a followed by the parameters (excepted i, ν and buf_{rec})

4.10 Gossiping subroutine

The gossiping[5] routine is frequently used in a lot of algorithms when all the processors have to broadcast the results of their previous computations to the other processors to reconstruct a global result on all the processors.

A call to the **loccs_ata** routine is realized by:

- loccs_ata(topo, ν, buf_{em}, buf_{rec}, size, job_b, nbpb, ..., job_a, nbpa, ...)

where:

topo	underlying topology
ν	the size of a packet
buf_{em}	emission buffer
buf_{rec}	reception buffer
size	total size of the message
job_b	computations to be done on the message before the communication
nbpb, ...	number of parameters of job_b followed by the parameters (excepted i, ν and buf_{em})
job_a	computations to be done on the message after the communication
nbpa, ...	number of parameters of job_a followed by the parameters (excepted i, ν and buf_{rec})

4.11 Personalized gossiping subroutine

A call to the **loccs_pata** routine is realized by:

- loccs_pata(topo, ν, buf_{em}, buf_{rec}, size, job_b, nbpb, ..., job_a, nbpa, ...)

where:

[5] all-to-all

topo	underlying topology
ν	the size of a packet
buf_{em}	emission buffer
buf_{rec}	reception buffer
size	total size of the message
job_b	computations to be done on the message before the communication
nbpb, ...	number of parameters of job_b followed by the parameters (excepted i, ν and buf_{em})
job_a	computations to be done on the message after the communication
nbpa, ...	number of parameters of job_a followed by the parameters (excepted i, ν and buf_{rec})

5 Applications of the subroutines

In this section, we present some applications of the previously presented subroutines.

5.1 Transposition sub-blocks subroutine

The transposition of a matrix stored by sub-blocks on a square two-dimensional torus requires only exchanges of messages between diagonally opposed processors and thus we can directly use the previous **loccs_exchange** routine on every diagonaly opposed processors [9] (see figure 5 for an implementation on a grid network of n nodes).

```
begin
  ⋮
  me = i*√n + j
  ot = j*√n + i
  if (i ≠ j) then
    loccs_exchange(grid, me,ot, ν, buf_em,buf_rec, m, job_b,0,job_a,0)
  endif
  ⋮
end
```

Figure 5: Parallel transposition algorithm on a grid of n nodes

The **loccs_exchange** routine is then useful if computations have to be done before and after the transposition. This is the case of the level 3 BLAS routines when they use transposed matrices (see section 5.3).

5.2 Transposition row subroutine

The transposition row subroutine is involved in an algorithm where the matrix is distributed by rows on the network. This operation is a special case of personalized all-to-all routine.

This operation has already been studied by Saad [20] and Johnsson [18] for the hypercube topology.

This is a direct utilization of the **loccs_pata** routine (see figure 6 for its use on a ring).

```
begin
  ⋮
  loccs_pata(ring, ν, buf_em,buf_rec, m, job_b,0,job_a,0)
  ⋮
end
```

Figure 6: Parallel transposition row algorithm on a ring of n nodes

5.3 Rank-2k updates of a symmetric matrix

This operation is one of the routines of the level 3 BLAS. This is the routine SYR2K. It computes the operation:

$$C = \alpha AB^T + \alpha BA^T + \beta C \text{ and } C = \alpha A^T B + \alpha B^T A + \beta C$$

This operation requires only matrix product (AB^T), matrix sum and matrix transposition ($T = (AB^T)^T$) with scaling with α and β if needed [14]. This last operation can be combined with the sum of matrix T [5]. There is no job before.

```
begin
  me = i*√n + j
  ot = j*√n + i
  load matrix A, B^T and C   (I)
  compute T = αAB^T          (II)
  compute C = βC + T         (III)
  transpose T                (IV)
  compute C = C + T^T        (V)
  unload matrix C            (VI)
end
```

Figure 7: Parallel Rank-2k algorithm

If the matrices are distributed by square sub-blocks on a mesh of processors, we use our loccs_exchange subroutine between diagonaly opposed processors to compute phases (IV) and (V) in the same call (figure 7). The job_{after} will be a sum of two sub-matrices [5] (see figure 8). SUM is the computation part for the diagonal processors.

5.4 LU factorization

LU factorization algorithm can be described with the row storage and row pivoting as in [16].

The LOCCS routine **loccs_ota** can be directly used for the LU factorization. On figure 9, the bold part can be replaced by a single call to the **loccs_ota** routine (in bold on figure 10).

Notice that, in the case of the LU factorization, there is no job_{after}. Therefore, NOJOB is given as a parameter instead of the routine job_{before}.

The resulting code is simpler and more efficient.

```
begin
  me = i*√n + j
  ot = j*√n + i
  load matrix A, B^T and C                                                    (I)
  compute T = αAB^T                                                           (II)
  compute C = βC + T                                                          (III)
  if i ≠ j then
    loccs_exchange(grid, me, ot, ν, T ,buf_rec, n*n, NOJOB ,sum,1, C)
  else
    SUM(C)
  unload matrix C                                                             (VI)
end
```

Figure 8: Parallel Rank-2k algorithm using loccs routine exchange

5.5 Cholesky factorization

Given a matrix A symmetric definite positive, we factorize A as the product LL^t where L is a lower triangular matrix.

If the matrix A is mapped on the nodes by columns, the parallel algorithm is given by figure 11 where *cmod(j,k)* computes column j as a function of all the previously computed columns k and *cdiv(j)* produce the final version of column j by dividing it by the square root of its diagonal element.

The version of the parallel algorithm for Cholesky factorization is straightforward using the **loccs_ota** routine, A[j] is the column j of matrix A, V is a temporary vector and n the size of the matrix. CMOD is the cmod operation on all the remaining columns of the processors (see figure 12).

5.6 Computation of the bidimensional DFT

The computation problem of the bidimensional DFT (Discrete Fourier Transform) is given by the equation [6, 21]:

$$X(j_1, j_2) = \sum_{k_1=0}^{n-1} (\sum_{k_2=0}^{n-1} x(k_1, k_2) * \exp^{(\frac{-i2\pi(k_1 j_1 + k_2 j_2)}{n})}) \qquad (1)$$

This equation can be transformed in order to obtain calls to the classical monodimensional FFT (Fast Fourier Transform (FFT_1D)) [6, 21].

The algorithm is then straightforward. Each processor receives a set of rows of the original matrix. And after having computed the FFT_1D on each row, a transposition is done on the matrix. After this communication operation, another FFT_1D is done on the resulting matrix (see figure 13 for the general algorithm).

The algorithm with a simple use of the LOCCS routine **loccs_pata** is given on figure 14. We overlap the transposition of the matrix by the computation of the monodimensional FFT of the remaining rows and the internal transposition of the received packets (job int_transp).

```
me = my_id()
for k=0 to n-1
   root = calc_root(k)
   determine pivot row
   update permutation vector
   /* computation part of the algorithm */
   if (me == root)
      /* I own pivot row */
      broadcast pivot row
   else
      receive pivot row
   /* update part of the algorithm */
   for (all rows  i > k that I own)
      l_ik = a_ik/a_k
      for j=k+1 to n-1
         a_ij = a_ij - l_ik * a_kj
      endfor
   endfor
endfor
```

Figure 9: Parallel LU factorization algorithm

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

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

```
me = my - id()
for k=0 to n-1
   ν = calc_nu_opt(k)
   root = calc_root(k)
   determine pivot row
   update permutation vector
   loccs_ota(topo,root, ν, A, A, n*n, NOJOB, update, 1)
endfor
```

Figure 10: Parallel LU factorization algorithm using LOCCS routine loccs_ota

```
for j=0 to n-1
  if col_j is one of my cols then cdiv(j)
  communicate(col_j)
  for all of my cols k > j do cmod(k,j)
endfor
```

Figure 11: Parallel Cholesky factorization algorithm

```
for j=0 to n-1
  loccs_ota(topo,root, ν, A[j], V, n, cdiv,0,CMOD,0)
endfor
```

Figure 12: Parallel Cholesky factorization algorithm using LOCCS routine loccs_ota

6 Conclusion and future work

The use of routines like the LOCCS prevents the developer of a parallel program from the redesign of a complicated pipeline program in order to increase the efficiency. Moreover, the code obtained is much simpler and more efficient. The routines can be optimized for each distributed memory machines and each topology. These routines correspond to the SPMD-like paradigm of execution.

Our current work is to implement this set of subroutines on the Intel iPSC/860, Paragon and ARCHIPEL Volvox IS-860 machines and to propose routines for the computation of ν in the classical algorithms. We investigate the possibility of a computation of ν during the execution to ensure a better load balance.

The LOCCS routines can also be called by a parallel compiler of FortranD-like programs which know the parameters of the target machine and the data dependences within the code.

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

Figure 13: General bidimensional FFT algorithm

```
begin
  loccs_pata(topo, ν, A, AT, n*n, FFT_1D, 1, n, int_transp, 0)
  Compute FFT_1D on resulting rows
end
```

Figure 14: Parallel bidimensional FFT algorithm using the LOCCS

References

[1] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. Van de geijn, *Blacs: Basic linear algebra communication subroutines*, in Sixth Distributed Memory Computing Conference, Portland, , U.S.A., 1991.

[2] ——, *Lapack for distributed memory architecture progress report*, in Fifth SIAM Conference on Parallel Processing for Scientific Computing,, U.S.A., 1991.

[3] C. Bischof, *Fundamental Linear Algebra Computations on High-Performances Computers*, tech. rep., Mathematics and Computer Science Division - Argonne National Laboratory, 1988.

[4] C. Bischof, J. Demmel, J. Dongarra, J. Croz, A. Greenbaum, S. Hammarling, and D. Sorensen, *LAPACK Working Note: Provisional Contents*, Tech. Rep. 5, Mathematics and Computer Science Division - Argonne National Laboratory, 1989.

[5] C. Bonello and F. Desprez, *Implementation of Linear Algebra and Communication Libraries on the TNode Reconfigurable Machine*, in Environments and Tools For Parallel Scientific Computing - St Hilaire du Touvet, Elsevier, September 1992.

[6] C. Calvin, *Benchmarks sur Machines Parallèles*, September 1992. Cnet Grenoble Report (in french).

[7] J. Choi, J. Dongarra, and D. Walker, *Level 3 BLAS for Distributed Memory Concurrent Computers*, in Environments and Tools For Parallel Scientific Computing, Elsevier, 1992.

[8] ——, *The Design of Scalable Software Libraries for Distributed Memory Concurrent Computers*, in Environments and Tools For Parallel Scientific Computing, Elsevier, 1992.

[9] F. Desprez and B. Tourancheau, *A Theoretical Study of Reconfigurability for Basic Communication Algorithms*, in CONPAR, 1992.

[10] J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transaction on Mathematical Software, 16 (1990), pp. 1–17.

[11] J. Dongarra and R. V. D. Geijn, *LAPACK Working Note: Two Dimensional Basic Linear Algebra Communication Subprograms*, Tech. Rep. 37, Department of Computer Science - University of Tennessee, 1991.

[12] J. Dongarra, R. V. D. Geijn, and R. Whaley, *Two Dimensional Basic Linear Algebra Communication Subprograms*, in Environments and Tools For Parallel Scientific Computing, Elsevier, September 1992.

[13] M. Dunigan, *Performance of the Intel iPSC/860 Hypercube*, tech. rep., Oak Ridge National Laboratory, September 1990.

[14] A. Elster, *Basic Matrix Subprograms for Distributed Memory Systems*, in Proceedings of DMCC5 - Charleston, IEEE Computer Society Press, 1990, pp. 311–316.

[15] G. Geist, M. Heath, B. Peyton, and P. Worley, *PICL: A Portable Instrumented Communication Library*, Tech. Rep. ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[16] G. Geist and C. Romine, *LU Factorization Algorithms on Distributed Memory Multiprocessor Architectures*, SIAM Journal on Science and Statistical Computing, 4 (1988), pp. 639–649.

[17] M. Heath and J. Etheridge, *Visualizing Performances of Parallel Programs*, Tech. Rep. ORNL/TM-11813, Oak Ridge National Laboratory, July 1991.

[18] S. Johnsson and C. Ho, *Optimum Broadcasting and Personalized Communication in Hypercubes*, IEEE Transaction on Computing, 8 (1989), pp. 1249–1268.

[19] M. Pourzandi and B. Tourancheau, *Recouvrement Calcul / Communication dans l'Elimination de Gauss sur iPSC/860*, Tech. Rep. 92-29, Laboratoire LIP, 1992.

[20] Y. Saad and M. Schultz, *Data Communication in Parallel Architectures*, Journal of Parallel and Distributed Computing, 6 (1989), pp. 115–135.

[21] O. Sentieys, H. Duboix, J. Philippe, and E. Martin, *A Methodologic Approach to Configure Architectures Applied to an MIMD Transputer Based Machine for Image and Signal Processing*, in Applications of Digital Signal Processing to Communications - Working Group 4 (WG4) Workshop on Massively Parallel Computing, Ecole Polytechnique Fédérale de Lausanne, March 1992.