Expanding the Message Passing Library Model with Nested Parallelism

Rodríguez C., Sande F., León C., García F.
Dpto. Estadística, Investigación Operativa y Computación,
Universidad de La Laguna, Tenerife, Spain

Abstract. A synchronous extension to the library model for message passing (Inmos C, PVM, Parmacs, MPI, etc.) is presented. This extension, provides a comfortable expression of nested parallelism from inside the message passing model. Furthermore of being a valuable tool for the presentation and teaching of parallel algorithms, the computational results prove that an efficiency similar to or even better than the one obtained designing and implementing algorithms using the native language can be achieved.

Keywords: Nested Parallelism, Multicomputers, Message Passing Libraries, inmosC, PVM, MPI.

1. Introduction.

Although the most safe approach to the general expression of parallel algorithms comes from the CSP model [18] and its most successful implementation, occam [21], the lack of portability has resulted in the translation of the message passing ideas to libraries extending sequential languages. In the parallel programming model based on a library for message passing (Inmos C [20], PVM [10], Parmacs [15], MPI [28], etc.) a collection of processes execute programs written in a standard sequential language, usually C or Fortran, extended with a library of functions for sending and receiving messages and the creation and synchronization of processes. From all the existing approaches, this one seems the most general, and the one leading to more efficient code. Nevertheless, the cost of writing parallel codes with these tools has resulted in an effort directed to the development of higher level languages being portable and efficient for a wide range of supercomputers. An usual approach has been to provide data-parallel constructs to existing sequential languages. This is the case of High Performance Fortran (HPF) [16] and different C extensions such as C* [34], [32] and C** [25]. Although the eclosion of HPF seems to have been the most successful, it is commonly agreed that still there are some inconvenients. The performance of a HPF program does not only depend on the skills of the programmer but also on the capacity of the compiler. Usually, the structure and consequently the performance of a program are neither intuitive nor obvious to the programmer [9]. Even simple operations on not aligned arrays may demand a considerable amount of communications if the arrays have different distributions. Although these languages are appropriate for matrices and regular meshes, they are not suitable for irregular structures such as sparse matrices, graphs and trees. According to Blelloch [4], this data parallelism is limited to its less expressive form: flat data-parallelism.
Languages with nested parallelism such as Paralation Lisp [6], NESL [4], fork95 [23], [11] and ll [35] provide the capacity to nest parallel calls. This model combines the programming ease of the data-parallel model with the efficiency on execution for irregular data structures of the

control-parallelism model [4]. In [6] we can find information about an implementation of the quicksort algorithm using Paralation Lisp that is only a factor of three times slower than the fastest sorting algorithm for the CM2. These results have been improved for NESL [5]. Nevertheless, NESL is far away of being a well known and extended language, perhaps due to its nature of functional language. In opposition, fork95 and *ll* are procedural languages based on C and *Pascal-* [14] respectively. The present implementation of the fork95 compiler produces code for a SBPRAM computer [1] based on the PRAM model [8]. At this moment, the machine is being built and there are no fork95 compilers for other parallel computers.

In this paper we describe a subset of *llc*, an extension of the library model for message passing that allows a comfortable expression of nested parallelism from inside the message passing model. The computational results show that the efficiency obtained is equivalent to the one reached designing the program in the native language for a concrete architecture and topology.

Through the use of two examples, the next section outlines the syntax and semantic of *llc*. The third section portrays some implementation issues for multicomputers. Computational results are presented in the fourth section. Conclusions and work under development are commented in the fifth and last section.

2. Description of the *llc* extension: Use and Semantics.

More than a formal approach, we will use a few examples to describe the use, syntax and semantic of the synchronous version of the *llc* system (the name starts with the two initials of La Laguna. Read it "La Laguna C"). The current version expands inmosC, but the system can be straightforwardly ported to any extension of C with a message passing library like PVM, MPI, MPL, etc. The *llc* environment provides the programmer with nested parallelism and, in consequence, with the capability to mix parallelism and recursivity. This feature makes easier the implementation of parallel divide and conquer algorithms [22]. The *llc* simple code in figure 1 shows a "natural" parallel version of the classical quicksort algorithm [19] sorting a global array *A*. Procedure *find* in figure 2, due to Hoare [17] divides the array *A* in two balanced halves in such a way that all the items in the second half are greater than all the elements in the first half. This is achieved by iterative calls to the well known procedure *partition* [19]. It can be easily proved that procedure *find* is twice slower than the classic procedure *partition* [13]. The code in figure 1 is almost self-explanatory. At the beginning of the computation all the processors belong to a single set and the variable NUMPROCESSORS contains the number of processors available in the machine. The word "set" has an special meaning in *llc* and it is used to describe a synchronous group of processors. This concept is pretty similar to the concept of group introduced in FORK [11] but, in contrast to FORK, conditionals and loops do not create implicit subsets. The *llc* macro PAR used in line 9 performs in parallel the two recursive calls to sort the first and second subintervals.

The additional second and third parameters, *A + first* and *size*, show that the result of the call to the function *qs(first, middle)* on the first segment is constituted by the size bytes pointed by *A + first*. In the same way, the two last parameters *A + middle + 1* and *size* describe the fact that the result of the second call is constituted by the size bytes following the address pointed by *A + middle + 1*. At the time to execute the macro PAR at line 9 in figure 1, the set is divided into two subsets. One executing the first call and the other doing the second one. The variable NUMPROCESSORS is updated for the two subsets according with the distribution policy (that policy can be chosen and/or modified by the programmer).

```
void qs(int first, int last) {
 int middle, size;

 if (NUMPROCESSORS > 1) {
   if (first < last) {
     middle = (first + last) / 2;
     size = (middle - first + 1) * sizeof(int);
     find(first, last, middle);
     PAR(qs(first, middle), A + first, size,
         qs(middle + 1, last), A + middle + 1, size)
   }
 }
 else {
   seqquicksort(first, last);
 }
} /* qs */
```

Figure 1: Parallel quicksort described in *llc*.

At the end of the calls, the two splitted subsets swap the results and they join together to form the original set. There are other macros allowing the creation of new subsets, with the same resemblance of PAR, like PARVIRTUAL (to support virtual processors), PARLEFT (instead of swapping the results, only processors in the first subset receive the results generated by the processors in the second subset), etc.

```
void find(int first, int last, int middle) {
  int left, right, i, j;

  left = first; right = last;
  while (left < right) {
    partition(&i, &j, left, right);
    if (middle<=j) right = j; else if (i<=middle) left = i;
    else left = right;
  }
} /* find */
```

Figure 2: Function *find*: balancing the partition.

To take advantage of the *llc* programming scheme the programmers have only to include in the body of their program the file llcsync.h:

#include <llcsync.h>

and to follow a reduced set of semantic rules. Generally, the *llc* programmer writes only one code, obviating the usual differentiation in message passing programs between "master" or "root" processor and "workers" or "slaves". Figure 3 contains the corresponding *llc main()* function. As it can be deduced from their names, macros INITIALIZE and EXIT, included in llcsync.h, are responsible for the respective execution of the initialization and finalization code of the *llc* system. Observe in line 11 the use of the *llc* function GPRINTF allowing to all the processors in the root subset to gain access to the output to write their NAME and computing time. The *llc* variable NAME holds the processor name.

```
main(void) {
  clock_t itime, ftime;
  int first, last;

  INITIALIZE;
  initialize(A);   /* Read array A */
  itime = clock();
  first = 0; last = SIZE - 1;
  qs(first, last);
  ftime = clock();
  GPRINTF("\n%d: time: (%lf)\n",
           NAME, difftime(ftime,itime));
  EXIT;
} /* main */
```

Figure 3: main() function: same code for both root and node processors.

Calls to INITIALIZE, EXIT and GPRINTF are examples of *subset operations*. The main rule to consider when using a *subset operations* in *llc* is:

RULE:

All the processors in a subset have to participate when executing a "subset operation".

It is an error, if some of the members of a subset do execute a *subset operation* and there are other members that do not. The *llc* manual describe what functions are *subset operations* and what are not. Any *subset operation* implies a synchronization among the members of the subset. Viceversa, these are the only synchronizations provided by the *llc* system. There are a variety of *subset operations*: reductions like REDUCEBYADD, REDUCEBYMULT, REDUCEBYMIN, REDUCEBYMAX and REDUCE (reduction using a function specified by the user) or communications like ONETOALL, ALLTOALL, PONETOALL (personalized broadcasting), PALLTOALL (personalized all to all communication), SHIFTLEFT, SHIFTRIGHT, etc. with meanings similar to those described in [24]. Even the PAR family of macros (PARVIRTUAL, PARLEFT, etc.) are *subset operations*. Obviously, the programmer can extend this range of operations by using the native language.

A second example of *llc* appears in figure 4. This code, when executed by a subset, searches for the first appearance of an element *KEY* in an array. This code works even if there has been previous parallel calls (like the macro PAR). The macro IAMTHELASTPROCESSOR in line 8 returns 1 (TRUE) for the last processor in a subset. After the call to the *subset operation* REDUCEBYMIN in lines 11-16, variable *i0* holds the minimum index *i* such that *array[i]* is equal to *KEY*. When called by a subset *S*, macro REDUCEBYMIN(x, y), sets all variables *y* to the minimun of the values *x* brought by the processors in *S*. This code is correct since the call is executed by all the processors in the subset. It would be wrong if the else clause is suppressed in lines 11-16 and substituted by:

```
if (i < last) {
        REDUCEBYMIN(i0, i)
}
```

```
ini_time = clock();

i0 = 0;
rate = SIZE / NUMPROCESSORS;
remaining = SIZE % NUMPROCESSORS;
first = rate * NAME;
last = first + rate;
if (IAMTHELASTPROCESSOR)
  last = SIZE;
for (i = first; ((i < last) && (array[i] != KEY)); i++);
if (i < last) {
  REDUCEBYMIN(i0, i)
}
else {
  REDUCEBYMIN(i0, INFINITY);
}

/* now for every processor in the subset it holds:  */
/*         i0 = min { i / array[i] == KEY}          */

end_time = clock();
GPRINTF("\n%d: time: (%lf)\n",
            NAME, difftime(end_time, ini_time));
if (i0 == INFINITY) GPRINTF("%d: NOT Found\n", NAME);
else GPRINTF("%d: Found Position: %d\n", NAME, i0);
```

Figure 4: Searching for the first ocurrence of KEY in an unsorted array.

3. Schemes for the efficient implementation on multicomputers.

All the processors are active at the beginning of the computation and they belong to the same synchronous set H. For each processor there are, among others, two variables involved in the division and rejoinment of subsets. These two variables are: NUMPROCESSORS and NAME. The *llc* programmer has read-only access to these variables. NUMPROCESSORS stands for the total number of processors in its subset H. All the processors in H have the same value for NUMPROCESSORS. Variable NAME, always with NAME between 0 and NUMPROCESSORS-1 alludes to the name of the processor inside the subset H. After the execution of a call to a PAR macro (or one of the variety of macros creating synchronous subsets), the set H is divided in the number r of synchronous subsets $H_1, H_2, ..., H_r$ requested by the particular call, following the policy specified by the programmer (CYCLIC, BLOCK, etc.). NUMPROCESSORS is updated to $/H_i/$ for each processor in subset H_i, $i = 1, ..., r$. Processors in H_i are reenumerated with NAMEs between 0 and $/H_i/ - 1$. As an example, for a hypercube computer, the call to the macro PAR at lines 9-10 of figure 1, using a BLOCK policy, expands to the code in figure 5.

```
 1  {
 2    BOOLEAN INLOWHYPERCUBE = ((NAME & (1 << BIT)) == 0);
 3
 4    PUSHPARALLELCONTEXT;
 5
 6    /* Subset division phase */
 7    NUMPROCESSORS /= 2;    /* Block policy */
 8    NAME /= 2;
 9    BIT --;
10    /* Swapping the results of the two parallel calls */
11    if (INLOWHYPERCUBE) {
12      qs(first, middle);
13      SWAP(BIT, A + first, size, A + middle + 1, size);
14    }
15    else {
16      qs(middle+1, last);
17      SWAP(BIT, A + middle + 1, size, A + first, size);
18    }
19    /* Rejoinment phase */
20    POPPARALLELCONTEXT;
21  }
```

Figure 5: Expansion of the PAR call.

The call in the main function to the macro INITIALIZE at line 5 of figure 3, initializes the variable BIT to the dimension of the hypercube. The macro PUSHPARALLELCONTEXT stores the three variables NUMPROCESSORS, NAME and BIT characterizing the current subset. The division of the original subset into two subsets according to the BLOCK policy is carried out in lines 7-9. The call to the macro SWAP produces an interchange in dimension BIT of the resulting data. The POPPARALLELCONTEXT macro joins the two subsets by recovering the stored context. In hypercubic networks, butterflies, perfect-shuffles, cycle connected hypercubes, etc. these three phases of organization, interchange and reorganization can be achieved efficiently. The hypercube implementation relies on the fact that, at any time, an *llc* subset of processors corresponds to a sub-hypercube of the initial hypercube.

4. Computational Results.

The results presented in this section were obtained executing the algorithms on a Parsys SN-1000 with T-800 transputers at 20Mhz and links operating at 10Mb/s. Experiments with 2, 4 and 8 transputer hypercubes were carried out. For each hypercube size and for each algorithm the corresponding entry shows the speedup against the corresponding best sequential algorithm. All the algorithms were coded using inmosC. Integer arrays of size 256K, 512K and 768K have been used as input.

Table I compares the efficiency of the *llc* algorithm in figure 1 (columns labelled *llc*) with the obtained programming in inmosC professor P.B. Hansen [12] parallel quicksort for hypercubes (columns labelled BH).

	2		4		8	
	BH	llc	BH	llc	BH	llc
256 K	0.79	1.16	1.04	1.78	1.21	2.23
512 K	0.74	1.06	0.97	1.61	1.13	2.03
768 K	0.81	1.13	1.08	1.73	1.27	2.17

Table I: Speedups: Hansen versus *llc* parallel quicksorts.

Previous experiences of the authors prove that the speedups obtained for the Hansen algorithm using inmosC are worse than using occam [3]. The results clearly prove the superiority of the *llc* implementation in spite of being easier to program.

The results in table II correspond to the *llc* implementation of the search algorithm in figure 4. Ten experiments were carried out. For each experiment, the position of the KEY was placed according to the uniform distribution between 0 and SIZE-1. Entries in table II show the average speedup for these ten cases. Although the synchronism of the parallel algorithm constitutes a drawback, the average speedup exhibits a good scalability.

	2	4	8
256 K	1.56	3.19	6.39
512 K	1.69	3.20	6.39
768 K	1.69	3.20	6.39

Table II: Search algorithm.

5. Conclusions and Future work.

A programming methodology and its corresponding tool have been presented. This tool consists in a set of C macros and functions expanding a programming environment based in the library model for message passing (Inmos C, PVM, Parmacs, MPI, etc.). Nested parallelism and the capacity to mix parallelism and recursivity are provided among other features. While portability across platforms can be improved, the programmer control is not diminished since the tool is embedded in the native C language. The sensible use of this tool does not introduce any kind of inefficiency.

Different versions of the library providing automatic load balancing (using dimension and diffusion techniques [33]), support for pipeline parallelism using the techniques explained in [2], [29] and support for asynchronous computing are being developed. The complexity of the expression of parallel probabilistic heuristics [31] (genetic algorithms, simulated annealing, tabu search, etc.), parallel divide and conquer algorithms [13], [12], [3], parallel dynamic programming [30] and parallel branch and bound algorithms [26], [27], [36], [7] with these libraries is almost reduced to its sequential expression.

All the codes appearing in this paper can be obtained through anonymous ftp at: ftp.csi.ull.es/pub/parallel/llcsync.

References:

[1] Abolhassan, J. Keller, J. Paul W.J. On physical Realizations of the Theoretical PRAM Model. *Technical Report 21/1990*, Universität des Saarlandes, SFB 124, 1990.

[2] Almeida F., García F., Morales D. and Rodríguez C.. A parallel algorithm for the integer knapsack problem for pipeline networks. *Journal of Parallel Algorithms and Applications* 6(3-4) (1994).

[3] Almeida F., Garcia F., Roda J., Morales D. and Rodríguez C.. A Comparative Study of Two Distributed Systems: PVM and Transputers. *Transputers Applications and Systems '95*. IOS Press (1995) 244-258.

[4] Blelloch, G. E., Hardwick J., Sipelstein J, Zagha M and Chatterjee S.. Implementation of a Portable nested Data-Parallel Language". *Journal of Parallel and Distributed Computing* 21 (1994) 4-14.

[5] Blelloch, G., and Greiner, J. A Provable Time and Space efficient implementation of NESL. *ACM Sigplan International Conference on Functional Programming* (1996) 213-225.

[6] Blelloch, G. and Sabot G., Compiling Collection-Oriented Languages onto Massively Parallel Computers. *Journal of Parallel and Distributed Computing* 8(2) (1990) 119-134.

[7] Burns, A. G., Friedman, F. Parallel Ordered Enumerative Algorithms for the Multidimensional Knapsack

Problem on Transputer Networks. *Transputer Research and Applications 7*. IOS Press. (1995) 116-129.
[8] Fortune S. Wyllie, J.. Parallelism in Random Access Machines. *STOC* (1978) 114-118.
[9] Foster I., Designing and Building Parallel Programs. Addison-Wesley. 1994.
[10] Geist A., Begelin A., Dongarra J., Jiang W., Mancheck R. and Sunderam V., PVM 3.1. User's Guide and Reference Manual. Oak Ridge Laboratory. 1993.
[11] Hagerup, T. Schmidt and Seidl, H. FORK: A High-Level Language for PRAMs, *Future Generation Computer Systems* 8 (1992) 379-393.
[12] Hansen P. B., Do hypercubes sort faster than tree machines?. *Concurrency: Practice andExperience* 6(2) (1994) 143-151.
[13] Hansen P. B., Studies in Computational Science. Parallel Programming Paradigms. Prentice Hall. 1995.
[14] Hansen, P. B., Brinch Hansen on Pascal Compilers. Prentice-Hall. 1985.
[15] Hempel R., Hoppe H., and Supalov A., PARMACS 6.0 Library Interface Specification. Techical Report, GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany (1992)
[16] High Performance Fortran Forum. High Performance Fortran Language Specifications, 1993.
[17] Hoare, C. A. R., Proof of a Program: Find. *Communications of the ACM* 14 39-45.
[18] Hoare, C. A. R., Communicating Sequential Processes, Prentice Hall International, 1985.
[19] Hoare, C. A. R. Algorithm 64: Quicksort. *Communications of the ACM*, 4, (1961) 321.
[20] INMOS Limited. ANSI C toolset reference manual, INMOS limited, 1990.
[21] INMOS Limited. Occam2 Reference Manual, Prentice Hall International Series in Computer Science, 1988.
[22] JáJá J., An Introduction to Parallel Algorithms, Addison-Wesley, 1992.
[23] Kessler, C., Seidl, H. Integrating Synchronous and Asynchronous Paradigms: The Fork965 Parallel Programming Language. *Proceedings of MPPM-95 Conference on Massively Parallel Programming Models.* IEEE CS Press, Berlin, 1995.
[24] Kumar, V. Grama, A., Gupta A., Karypis, G., Introduction to parallel Computing: Design and Analysis of Algorithms. The Benjamin/Cummings Pub. Co. 1994.
[25] Larus, J.R., Richards, B., and Viswanathan, G., C**: A large-grain, object oriented, data-parallel programming language. *Tech. Rep. UW #1126*, Computer-Science Dep. University of Wisconsin-Madison, Nov. 1992.
[26] Lüling R. and Monien B.. Two strategies for solving the vertex cover problem on a transputer network. *3rd International Workshop on Distributed Algorithms*, LNCS392, (1989) 160-171.
[27] Mckeown G., Rayward-Smith V., Rush A. and Turpin H., Using a transputer network to solve branch-and-bound problems. *Proceedings of the TRANSPUTING '91 Conference*, IOS Press. (1991) 781-800.
[28] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications and High Performance Computing*. 8 (3/4) (1994).
[29] Moldovan D. and Fortes J.. Partitioning and Mapping Algorithms into fixed size Systolic arrays, *IEEE Trans. Comput. C-35* (1) (1986) 1-12.
[30] Morales D., Roda J., Almeida F., Rodríguez C. and García F.. *Integral Knapsack Problems: Parallel Algorithms and their Implementations on Distributed Systems*. *Proceedings of the 1995 International Conference on Supercomputing*. ACM Press. (1995) 218-226.
[31] Moreno J.A., Roda J.L. y Moreno-Vega J.M., Parallel Genetic Algorithm for the discrete p-median problem. *Studies on Locational Analysis* (7) (1995) 131-141.
[32] Quinn, M.J. and Hatcher, P.J. Data Parallel programming on multicomputers *IEEE Software* 7(5) (1990).
[33] Rodríguez C., González D., Almeida F., Roda J., García F., Parallel Algorithms for Polyadic Problems. *5th EUROMICRO Workshop on Parallel and Distributed Processing (PDP'97)*. London. Enero 1997.
[34] Rose, J.R. and Steele, Jr., G.L., C*: An extended C language for data parallel programming. *Proceedings, Second International Conference on supercomputing* 2 (1987) 2-16.
[35] Sande, F., García, F., León, C., and Rodríguez, C.. The ll parallel programming System. *IEEE Transactions on Education* 39 (4), (1996) 457-464.
[36] Troya J. and Ortega M.. A study of parallel branch-and-bound algorithms with best-bound-first search. *Parallel Computing* (11) (1989) 121-126.