

Towards Structured Parallel Programming

Antonio J. Dorta, Jesús A. González, Casiano Rodríguez and Francisco de Sande

Dpto. Estadística, I.O. y Computación
Universidad de La Laguna
La Laguna, 38271, Spain
casiano@ull.es

Abstract. This paper presents part of our efforts to design a language giving support to the most common used programming patterns on distributed networks: farms, pipelines, data parallel loops and reductions. The language follows the OpenMP syntax wherever there is one for that pattern. In most cases, the OpenMP syntax is extended with additional annotations. As a consequence, the parallelisation process can follow a progressive approach. From the sequential program and through the use of OpenMP directives we can produce a new parallel shared memory version. From the OpenMP program and through the addition of supplementary distributed memory directives, we also can, without destroying the OpenMP program, produce a new parallel distributed memory version. If this approach fails, the efficiency of the program can be improved using a mix of the proposed skeletons with explicit MPI send/receive operations. The computational results corroborate the feasibility of the approach.

1 Introduction

It is not easy to program distributed memory parallel computers concisely to make them work efficiently. To alleviate this situation, some authors have proposed an “Skeleton based” approach [5, 9]. Skeletons are software components that reflect the common patterns of most parallel programs. A “good skeleton” is a carefully defined, reusable, parameterised programming form with pre-packaged implementations for different parallel architectures. The gist idea is that useful patterns of parallel computation and interaction can be packaged up as “second order” constructs (i.e. parameterised by other pieces of code), perhaps presented with reference to explicit parallelism, perhaps not. Implementations and analyses can be shared between instances. Such constructs are “skeletons”, in that they have structure but lack detail. The goal of the skeleton approach is to develop a viable and formally well-founded methodology of parallel programming based on a restricted set of frames/structures, avoiding when possible the send/receive mechanism of conventional parallel programming. Dijkstra structured programming, with the inclusion of the for, while, repeat, etc skeletons, rejecting the use of unstructured gotos is an analogy in the scope of sequential programming.

Skeletons allow both the expression of task parallelism (exploited among independent unrelated tasks) and data parallelism (exploited within the computation of a single task). Moreover there is a need for skeletons allowing the composition of other skeletons. Although there are many flavours of parallel skeletons, it is clear the importance of these:

- FARM/WorkQueuing, modelling a set of identical workers computing in parallel a stream of independent tasks.
- PIPE, exploiting pipeline parallelism in the evaluation of a cascade of stages.
- MAP/forall, modelling independent data parallel computations in which the same function is applied to all the elements of a data array.
- REDUCE, SCAN implementing parallel reduction and prefix computations of the elements of an array by means of an associative and commutative operator.

The names are obviously arbitrary; the ones here follow [9]. As with their sequential counterparts, a good parallel skeleton oriented model has not only to consider these but to allow the efficient nested combination of any of them: one process inside the execution of a PIPE may want to call a MAP that itself calls a REDUCE etc.

This paper presents part of our efforts to design a language giving support to the aforementioned skeletons. Although not integrated in the whole system, a stand-alone PIPE skeleton is already developed [7]. Work on the FARM/WorkQueuing skeleton is in progress [1]. Other authors have also proved the feasibility of implementing this skeleton on distributed machines [4]. This paper concentrates on the nesting of MAP/forall and reduction skeletons.

Section 2 contains an introduction to the syntax and semantic issues. Instead of choosing a new syntax we have preferred to follow the OpenMP syntax wherever there is one for that skeleton. On the shared memory programming area, OpenMP has landed as a new standard. It comes with support for the MAP/forall skeleton (expressed through the `for pragma`), for a relatively limited number of reductions and collective operations (MPI support in this area is by far richer) and, although not included in the standard, the KAI group has proposed and implemented extensions for the FARM/WorkQueuing skeleton [11].

The skeletons extend the OpenMP directives with new annotations. From the programmer perspective, it means that parallelisation follows the progressive path proposed by OpenMP. From the sequential program and through the use of OpenMP directives we can, without destroying the sequential program, produce a new parallel shared memory version. From the OpenMP program and through the addition of supplementary distributed memory directives, we also can, without destroying the OpenMP program, produce a new parallel distributed memory version. Still, if the efficiency is unsatisfactory, the programmer can mix the use of the proposed skeletons with explicit MPI send/receive operations.

Section 3 introduces the MAP/forall skeleton, using two well known examples. The syntax, semantic and some implementation issues related with the

pipeline skeleton are discussed in section 4. Section 5 deals with the nesting of skeletons. The computational results for the data parallel examples are presented in section 6. Computational results for the pipeline skeleton can be found in [7].

2 The Computing Model

To explain the proposal we will introduce the *One Thread is One Set of Processors* model, abbreviated *OTOSP*. This (ideal) model facilitates the interpretation of how we intend to map the skeletons on a distributed memory machine.

Let us imagine a machine composed of an *infinite* number of processors, each one with its own private memory and a network interface connecting them. The processors are organised in *sets*. At any time, the memory state of all the processors in the same set is identical. An *OTOSP* computation assumes that all the processors in the same set have the *same input data* and *the same program in memory*. The initial set is composed of all the processors in the machine. The only difference among the processors is an internal register, containing an integer, the *NAME (number)* of the processor. Consider the code in figure 1. Initially, the infinite processors are in the same set, represented by the root of

```
1 #pragma omp parallel for
2 #pragma llc result(ri+i, si[i]);
3 for(i=1; i<=3; i++) {
4   ...
5   #pragma omp parallel for
6   #pragma llc result(rj+j, sj[j]);
7   for(j=0; j<=i; j++) {
8     rj[j] = J_function(i, j, &sj[j], ...);
9   }
10  ri[i] = I_function(i, &si[i], ...);
11 }
```

Fig. 1. Two nested for statements

the tree in figure 2. All the processors are executing the same (sequential) thread and have identical values stored in their local memory. The `parallel for` in line 1 divides the set in three *subsets*.

After the execution of the loop, and to keep the coherence of the memory, each processor exchanges with its two *neighbours* the corresponding results. This is the meaning of the `result` directive at line 2. To inform the compiler about the data that has been modified inside each iteration of the `for`. Its argument is a list of pairs (*pointer expression, size*) specifying which variables have been changed during iteration *i* of the loop.

Thus, processor 0 in the first set/thread executes iteration `i=1` and at the end of it, it sends `si[1]` bytes starting at the address pointed by `ri+1` to processors 1

and 2 (across lines at the first level of the tree depicted in figure 2). Furthermore, it receives `si[2]` bytes from processor 1 and stores these bytes in its local memory starting at address `ri+2`. Analogously, receives `si[3]` bytes from processor 2 starting at the address pointed by `ri+3`. The same exchange is repeated among the other corresponding triplets (3, 4, 5), (6, 7, 8), ... You can easily visualise the operation if you realize that any new nested *for* creates/structures the current subset according as a *M-ary* hypercube, where *M* is the number of iterations in the parallel loop. Thus, this first *for* produces “the face” of a ternary hypercubic dimension, where every corner has two neighbours.

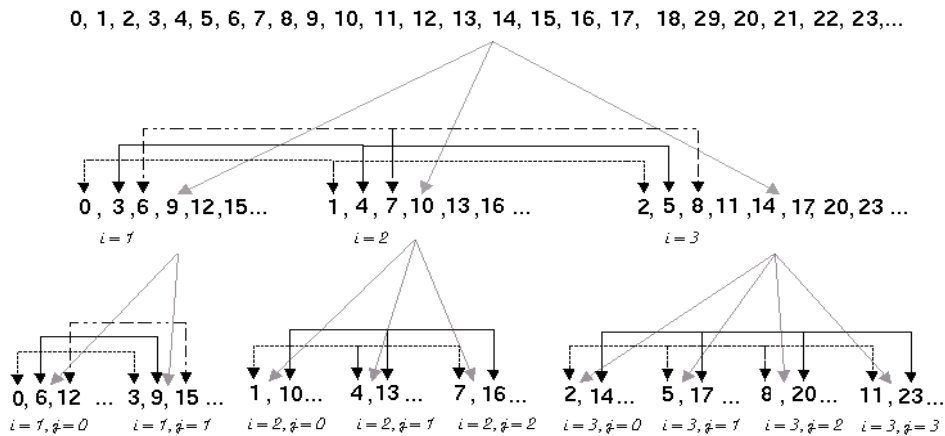


Fig. 2. The mapping associated with the two nested *for*s in figure 1

The second nested *for* at line 5 requests for different number of threads by the different threads/sets. The first thread/set ($i = 1$) executes a loop of size 2, and so the set is divided following a binary hypercube. The loop for the second set ($i = 2$) is of size 3, and the processors in this subset are accordingly divided in a ternary dimension. The last set ($i = 3$) executes a *for* of size 4, and consequently the set is partitioned in 4 subsets. In this 4-ary dimension, each processor is connected with 3 neighbours in the other subsets. Therefore, at the end of the nested compound statement, processor 17 will send `rj+1` to processors 14, 20 and 23 and will receive from them `rj+0`, `rj+2` and `rj+3`. The same will occur with any of the quartets involved (across lines at the deepest level of the tree depicted in figure 2).

Observe how the behaviour of OpenMP for shared variables is “mimicked” through the additional information provided by the `result` directive.

An OpenMP compiler producing code for a shared memory machine simply ignores the `result` directive and delivers a semantically equivalent program.

Unfortunately, an infinite machine, as the one described above, is only an idealisation. Real machines have a restricted number of processors. This fact complicates the actual mapping and introduces new problems, load balancing being the most important. For the sake of simplicity, these issues will not be discussed here.

3 Data Parallel Skeletons

Probably the canonical first example of any Parallel API is the “computing π ” program. Let us follow the traditions. The code in figure 3 shows the calculus of π using the proposed language, *llc*. When compiled by an OpenMP compiler, the pragma `omp parallel for` in line 5 indicates that the iterations of the loop at line 7 can be splitted among the threads in the team. A block schedule is used as default policy. In OpenMP the clause `private` forces the allocation of local-storage for the variable `t`, so each processor works with its own local variable. The clause `reduction` ensures that each processor works with a private local-storage variable `pi` and that a reduction operation occurs at the end of the loop, so the local values are summed on a shared global-storage object.

```
1 double t, pi=0.0, w;  
2 long i, n = 100000000;  
3 double local, pi = 0.0, w = 1.0 / n;  
4 ...  
5 #pragma omp parallel for reduction(+:pi) private(t)  
6 #pragma llc reduction_type(double)  
7 for(i = 0; i < n; i++) {  
8     t = (i + 0.5) * w;  
9     pi += 4.0/(1.0 + t*t);  
10 }  
11 pi *= w;  
12 ...
```

Fig. 3. Implementation of π generator using *llc*

When compiled by *llc*, and as it occurs in the shared memory case, the loop iterations are splitted among the processors. In fact, the clause `private` is kept only for compatibility with OpenMP, since it is not required: all the storages are private. The OpenMP clause `reduction` indicates that all the local values of variable `pi` has to be added at the end of the loop. This operation implies a collective communication among all processors in the group and the updating of the variable with the result of the reduction operation. Since type analysis has not been included in the current prototype of the compiler, the type of the

reduction variables has to be specified. This is the purpose of the (unnecessary and therefore to be eliminated in future versions) `pragma llc reduction_type` in line 6.

```

1 void compute(int np, int nd, double *box, vnd_t *pos, vnd_t *vel,
2             double mass, vnd_t *f, double *pot_p, double *kin_p) {
3     double x, d, pot, kin;
4     int i, j, k;
5     vnd_t rij;
6
7     pot = kin = 0.0;
8
9     #pragma omp parallel for default(shared)
10    private(i, j, k, rij, d) reduction(+ : pot, kin)
11    #pragma llc reduction_type (double, double)
12    #pragma llc result(f[i], nd)
13    for (i = 0; i < np; i++) { /* potential energy and forces */
14        for (j = 0; j < nd; j++)
15            f[i][j] = 0.0;
16        for (j = 0; j < np; j++) {
17            if (i != j) {
18                d = dist(nd, box, pos[i], pos[j], rij);
19                pot = pot + 0.5 * v(d);
20                for (k = 0; k < nd; k++) {
21                    f[i][k] = f[i][k] - rij[k] * dv(d) / d;
22                }
23            }
24        }
25        kin = kin + dotr8(nd, vel[i], vel[j]); /* kinetic energy */
26    }
27    *pot_p = pot;
28    *kin_p = kin;
29 }

```

Fig. 4. The Molecular Dynamics code in *llc*

Reduction and result clauses can be combined as shown in figure 4. This routine computes the forces and energies, given positions, masses, and velocities of `np` particles corresponding to the Verlet algorithm [13, 14]. The code is a *llc*-version of the fortran Molecular Dynamics (MD) code that can be obtained at the official OpenMP web site [8].

4 Pipeline Skeletons

Figure 5 illustrates the use of pipeline directives. It shows a *llc* implementation of a Dynamic Programming algorithm solving the Single Resource Allocation Problem (SRAP) [6]. This optimization problem consists on finding the optimal allocation of M units of an indivisible resource among N demanding tasks. Function $f_n(r)$ gives the profit obtained by the assignation of r units of resource to the n -th task. Using Dynamic Programming, the SRAP problem can be reduced to compute $G_{N-1}(M)$ using the equations: $G_n(r) = \max\{G_{n-1}(r-i) + f_n(i) : i \in \{0 \dots r\}\}$. The code `srap` takes as input the parameters N , M and the profit function $f(n, r)$. It fills in tables $G[n][r]$ and $L[n][r]$ with, respectively, the best profits and the optimal decisions.

```
1 int srap(int N, int M, cost f, table G, table L) {
2     int r, n, i, s, decision_i, temp, pos, chunksize, buffersize;
3
4     #pragma llc pipeline schedule(chunksize, buffersize)
5     #pragma llc result (&G[n][0], M) (&L[n][0], M)
6     for (n = 0; n < N; n++) {
7         if (n == 0)
8             for (r = 0; r <= M; r++) {
9                 G[0][r] = f(0, r); /* assume f is non decreasing */
10    #pragma llc send (&G[0][r], 1)
11    }
12    else
13        for (r = 0; r <= M; r++) {
14    #pragma llc receive (&G[n-1][r], &s)
15        temp = G[n-1][r];
16        pos = 0;
17        for (i = 1; i <= r; i++) {
18            decision_i = G[n-1][r-i] + f(n, i);
19            if (decision_i > temp) {
20                temp = decision_i;
21                pos = i;
22            }
23        }
24        G[n][r] = temp;
25    #pragma llc send (&G[n][r], 1)
26        L[n][r] = pos;
27    }
28    }
29    return G[N-1][M];
30 }
```

Fig. 5. Implementation of SRAP algorithm using *llc*

The pipeline directive used in line 4 has the following syntax:

```
#pragma llc pipeline [schedule(chunksz, buffsz)] [weight w]  
  for-loop
```

such directive indicates that the processors in the current subset (the word subset is used here in the sense explained in section 2) will be organized in a pipeline with the same number of stages N than iterations has the associated loop at line 6.

Each subset replicates the execution of the code associated with a loop iteration n . The use of the `result` and `reduction` clauses at line 5, ensures memory consistency at the end of the `pipeline`.

The initial stage, task `n==0`, is coded from lines 7 to 11. Assuming that the profit function is non decreasing, the optimal decision for any possible value `r` of resource is given by `f(0,r)`. Directive `send` at line 10 inserts an element in the data stream to the next stage. Communication occurs between consecutive subsets. The directive has no effect for processors in the last stage.

The rest of the stages perform the computation from lines 13 to 27. At line 14, directive `receive` takes the values from the previous stage stream, and stores them in `G[n-1][r]`. Value stored on variable `s` is discarded and must be the constant `sizeof(G[n][r])`. After computing a new value `G[n][r]`, it is sent to the next stage at line 25.

At the end of the computation, the n -th thread has computed the n -th column of matrices `G` and `L`. Directive `result` at line 5 forces the memory synchronization of all processors in the current group.

Implementation on a real parallel machine with a finite number of processors presents new problems. In particular the number of processors in each subset can be different and again, load balancing becomes an important issue. The clause `weight(w)` specifies an expression w that controls the number of processors assigned to each stage. The neighborhood relationship is redefined to include the situation when a processor has several neighbors in the next stage.

A second case corresponds to the usual situation when the number of stages is larger than the number of processors in the current set. Each subset is composed of a single processor and several stages have to be mapped onto it. In this case, the load balancing and the locality of data have to be considered. The clause `scheduled(chunksize, buffsize)` cyclically maps `chunksize` consecutive iterations in the same physical processors and buffers messages using buffers of size `buffsize`. This information allows the `llc` compiler to produce "tiled" code. The impact of tiling in the resulting performance can be seen in [7].

Observe that removing the `llc` directives from code `srap` leads to a valid sequential-C program.

5 Nesting the Skeletons

Arbitrary nesting of pipelines and for directives is allowed: a pipeline stage could contain a `forall` construct, and reciprocally, a `forall` iteration could execute a `pipeline` construct.

Figure 6 presents the *llc* version of the Fast Fourier Transform (*FFT*) algorithm [3]. The algorithm receives as input a pointer *a* to the original signal, the number *N* of points of the signal, a vector *W* containing the roots of unity and a value that indicates the *stride* between elements to be considered in the division stage of the algorithm. The function returns a pointer *A* to the transformed signal.

```

1 void FFT(Complex *A, Complex *a, Complex *W, unsigned N,
2         unsigned stride, Complex *D) {
3     Complex *B, *C;
4     Complex Aux, *pW;
5     unsigned i, n;
6
7     if(N == 1) {
8         A[0].re = a[0].re;
9         A[0].im = a[0].im;
10    }
11    else {
12        n = (N>>1);
13        B = D;
14        C = D + n;
15
16        #pragma omp parallel for
17        #pragma llc result(D+i*n, n)
18        for(i = 0; i <= 1; i++)
19            FFT(D+i*n, a+i*stride, W, n, stride<<1, A+i*n);
20
21        for(i = 0, pW = W; i < n; i++, pW += stride) {
22            Aux.re = pW->re * C[i].re - pW->im * C[i].im;
23            Aux.im = pW->re * C[i].im + pW->im * C[i].re;
24            A[i].re = B[i].re + Aux.re;
25            A[i].im = B[i].im + Aux.im;
26            A[i+n].re = B[i].re - Aux.re;
27            A[i+n].im = B[i].im - Aux.im;
28        }
29    }
30 }

```

Fig. 6. Implementation of FFT algorithm using *llc*

Parameter *D* is an auxiliary vector used in the combination stage. The execution of the code in figure 6 is based in the nesting of `parallel for` statements. Although nested parallelism is included in the OpenMP API [8], most current OpenMP compilers for shared memory machines serialise nested parallel loops. There are a few exceptions to this behavior [11, 12, 2].

Since it is an essential condition of the “Skeleton project” [9], *llc* allows the nesting of MAP/forall skeletons. The pragma `omp parallel for` at line 16 produces a partition of the current group of processors. A binary partition in two groups of the same size is performed to process the odd and even terms of the input signal separately. The pragma `llc result` at line 17 ensures a coherent view of the memory areas that have been modified in each subgroup.

When load balancing is an issue, a fair division of the group in equal size subgroups is inadequate. What is required in these cases is to expand the work sharing construct with a clause specifying either the group size or the weight associated with the corresponding work unit (see [10]).

6 Computational Results

To test the performance of the current *llc* prototype, the three previous examples in figures 3, 4 and 6 were run on a Cray T3E. The results are compared with carefully optimised MPI versions of these codes.

Figure 7 presents the speedup curves. The code in figure 3 and the corresponding MPI version, respectively labelled *pi_llc* and *pi_MPI*, were run with the parameter value $N = 10^8$. Both curves shown linear speedup.

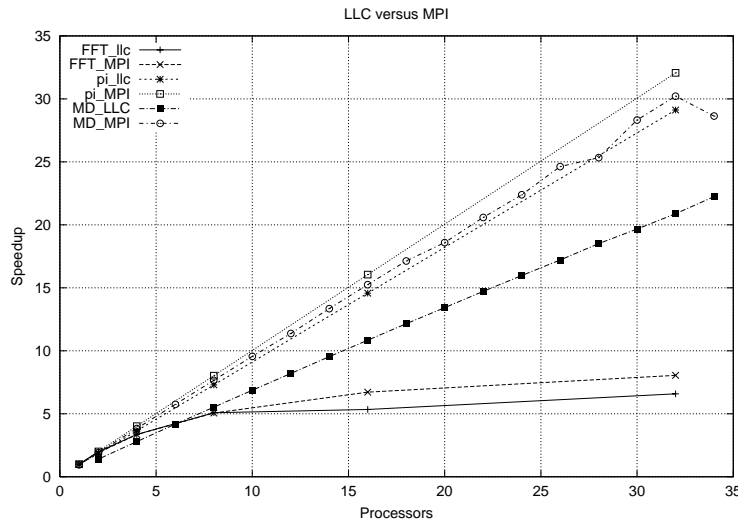


Fig. 7. *llc* versus MPI on a CRAY T3E

The number of particles for the MD algorithm was 8192. There were 10 simulation steps. The input signal for the FFT had 2^{20} complex points. Although *llc* and MPI curves show comparable behaviours, there is a larger separation. The reason being that the current prototype does not optimize the communications.

Acknowledgements

Thanks to the EPCC, CIEMAT, CEPBA, the EC TRACS program and SEUI. This work is partially supported by the Spanish MCyT project MaLLBa TIC1999-0754-C03.

References

1. Alba, E., Almeida, F., Blesa, M., Cabeza, J., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., León, C., Luna, J., Moreno, L., Petit, J., Rojas, A., Xhafa, F.: MALLBA: A Library of skeletons for combinatorial optimisation. Euro-Par 2002
2. Ayguade E., Martorell X., Labarta J., Gonzalez M. and Navarro N. Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study Proc. of the 1999 International Conference on Parallel Processing, Aizu (Japan), September 1999.
3. Cooley, J. W. and Tukey, J. W. An Algorithm for the Machine Calculation of Complex Fourier Series, *Mathematics of Computation*, series 19, 90, pp. 297–301. 1965.
4. Supercomputing Technologies Group. MIT Laboratory for Computer Science. The Cilk Project. <http://supertech.lcs.mit.edu/cilk>
5. M. Cole. *Algorithmic Skeletons: Structure Management of Parallel Computations*. MIT Press. 1989.
6. Morales, D., Almeida F., García F., Roda J.L., Rodriguez C. Design of Parallel Algorithms for the Single Resource Allocation Problem. *European Journal of Operational Research*. 126. pp. 166-174. 2000.
7. Moreno L.M., Almeida F., González-Morales D., Rodríguez C. Adaptive Execution of Pipelines. *LNCS. PVM/MPI*. 217-224. 2001
8. OpenMP Architecture Review Board: OpenMP C and C++ application program interface v. 1.0. <http://www.openmp.org/specs/mp-documents/cspec10.ps> October 1998.
9. Pelagatti S. *Structured Development of Parallel Programs*. Taylor and Francis. November 1998.
10. Rodríguez C., de Sande F., León C., García L. *Parallelism and Recursion in Message Passing Libraries: An Efficient Methodology*. *Concurrency: Practice and Experience*. 1999. Addison Wesley.
11. Shah S, Haab G, Petersen P, Throop J. Flexible control structures for parallelism in OpenMP. 1st European Workshop on OpenMP, Lund, Sweden, September 1999.
12. Tanaka Y., Taura K., Sato M., and Yonezawa A. Performance Evaluation of OpenMP Applications with Nested Parallelism. *Languages, Compilers, and Run-Time Systems for Scalable Computers* pp. 100-112, 2000
13. W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson. *J. Chem. Phys.* 76, 637 (1982)
14. M. Tuckerman, B. J. Berne, and G. J. Martyna, *J. Chem. Phys.* 97, 1990 (1992)