Efficient Use of Craft in Parallel Algorithms.

Joaquim Casulleras, Dept. de Fisica y Enginyeria Nuclear,
Universitat Politècnica de Catalunya-Cesca, Barcelona, Spain

1 Introduction.

Today's most important scientific problems require impressive and continuously increasing computing speeds. The use of massively parallel architectures appears to offer a unique possibility to achieve the computing resources needed at an affordable cost. However, coupled to the hardware development of new architectures, it arises the problem of adapting existing codes or developing new ones for the scientific problems that require the use of parallel computers. Furthermore, since the use of parallel computers will certainly become much common in a near future, not only the most computer demanding scientific problems, but practically all disciplines in which computing plays and important role will progress in the sense of using parallel algorithms. Thus, parallel programming tools which are not only efficient but also practical and easy to use are likely to gain a widespread use in a near future. The question of how the two virtues can better coexist in a single tool, and which should be the main characteristics of such a tool is therefore acute.

As a final user in writing scientific codes using different parallel computers and programming tools, such as the Occam language and the parallel CM Fortran90 in systems other than Cray, and the MPP programing model for the Cray T3D, my personal point of view about the usefulness of a programming model is that the significance of the facility of use, debugging and evolution of the parallel code is reinforced in front of the efficience. To put in other words: it is not difficult to have a tool that provides high efficiencies; what makes the great difference between different efficient tools is its facility of use. In this sense, the Craft programming model is remarkable. At first sight, it could seem to be a programming model conceived for the non experienced user, with reasonable chances of success in problems with some specific and somehow rigid parallelism. It's main usefulness would be the small changes required to transform a serial program into a parallel one, thus allowing for a smooth transition to the philosophy of parallel programming. On the other hand, it seemed to me less clear whether this programming model could successfully and efficiently afford a general parallel problem. Its operation in some real examples may give some insight with respect to this point.

Two different programming models, Craft and explicit message passing have been used for two different problems: Diffusion Monte Carlo calculations, and Molecular Dynamic simulations. The behaviour of the Craft code, compared to the one using explicit message passing calls, is analyzed. It will be shown that, contrary to the first impression, the Craft programming model is sufficiently flexible to handle appropriately almost any parallel problem.

2 Diffusion Monte Carlo calculations.

Monte Carlo methods have achieved in the last years a high level of accuracy in the description of atomic ^{3}He and ^{4}He, which constitute the most characteristics examples of quantum liquids. In the simulation of such systems, Diffusion Monte Carlo (DMC) algorithms solve stochastically the Schrodinger equation in imaginary time, using an iterative method that simulates an evolution over a small time Δt per iteration. The translation of the Schrodinger equation in terms of a stochastic algorithm defines the operations that have to be performed on the system. From an algorithmic point of view, it implies having a large number ($\approx$500) of different copies of a classical liquid, containing something like one hundred atoms each copy. The behavior of the different copies is dictated by the action of the quantum operators associated to the kinetic and potential energy, leading to a superposition of a deterministic and a random change in the positions of the atoms, plus an overall probability of the whole copy to replicate or dye.

The structure of the calculation therefore is decomposed in separate parts:

- Local evolution of single configurations, containing a deterministic part as well as a random one.
- Replication or death of configurations, depending on the "quality" of the random component of the algorithm for a particular configuration, compared to the average one.

The parallization of a computation of this type is naturally realized distributing the different configurations on the available processor elements. A parallel DMC computation using explicit message passing will be highly efficient, provided that a proper dynamic load balancing is achieved after the random replication or death of configurations. This will require a random pattern of communication similar to the one shown in Figure 1:

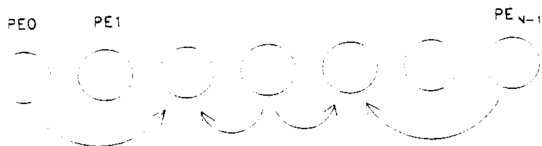

Figure 1

A calculation of this type enters naturally in the message passing programming model, and a PVM-like code is expected to show a fully satisfactory performance for this problem since

1. All operations are inherently local.
2. The user has full control over the flow of data, thus allowing for a precise load balancing, and minimum inter-processor communication.

One would expect then the efficiency of the message passing program to be very close to the ideal. This expectation becomes fully confirmed when we plot the speed-up attained using different numbers of processors, shown in Figure 2:

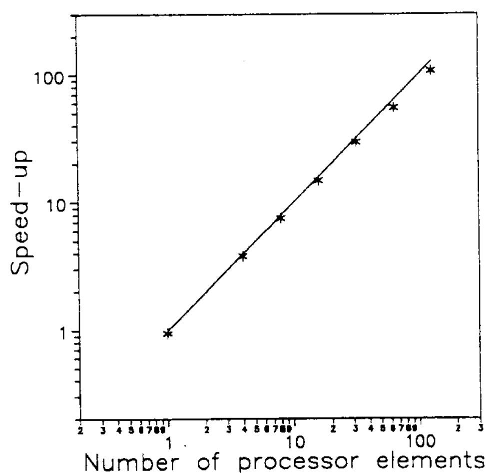

Figure 2

Now, let us consider how a Craft code could do in a calculation of this kind. There are two major issues to be considered: a) locality versus inter-processor communication, and b) load balancing.

Related to a) , one of the characteristics of Craft is that local operations which do not require inter-processor communication may originate remote read or write operations, if the compiler is not sure of it at compile time. However, in this example, the data distribution fits exactly with the *doshared* loop over configurations and the compiler does not succumb to this pitfall. With respect to b) , the fulfillment of a general inter-processor communication of the kind shown in Figure 01 is tailored with a minimal modification of a serial code. The Craft code could look like the program in the next column.

It is clear that the two points just considered are fully satisfied in the Craft approach, since the compiler can recognize at compile-time that all operations are local, and on the other hand the irregular pattern of communication needed can be fulfilled. One then expects a behavior similar to the one reached with

```
          DIMENSION   (:,:, ... , :)
CDIR$SHARED  x(:,:,  ...  ,  :block(1)

c         Local evolution of npop confi
CDIR$DOSHARED  ipop  ON  x(1,1...,
          do ipop = 1, npop
                 x(:,:, ... , in, ipop) =
            =          f ( x(:,:,  ... , in, ipop
          end do

c         Algorithm to build a new list
c         configurations taking into ac
c         the replication or death of th
c         ones:

          Build   nfuture(:)

c         Inter-processor   communicati
CDIR$DOSHARED  ipop  ON  x(1,1...,
          do ipop = 1, npop
                 x(:,:,  ...  ,  io,  nfuture(ipo
            =         x(:,:,  ... , in, ipop)
          end do
```

explicit message passing. The comparison of the efficiency is shown in Figure 2. The behavior of the two approaches is fully satisfactory in both cases, the unobservable differences between the two being produced only by the different rates of data transmission. Now, one could figure that the success of the Craft approach in this example is merely due to the simplicity, in terms of parallelization, of the DMC algorithms. To gain clarity on this point, let's consider another relevant problem, more entangling from the point of view of parallelization.

3 Molecular Dynamics Simulations

Molecular Dynamics (MD) simulations constitute a broad field of research, suitable for the study of statistical mechanical systems, ranging, say, from simple metal liquids to macro-molecules in environments of biological interest. Due to the very different systems that can be studied, the size and complexity of the "sample" to be studied by computer simulation may vary substantially. Typically 100 to 10,000 particles are used to obtain relevant information. In order to readily illustrate the kind of parallel algorithms that can be used, we will focus on the particular type of simple model interactions described by long range pair interactions. In this particular problem, the heart of the calculation would consist in long sequence of elementary updates of the system, each one involving three basic steps:

(i) Evaluation of the forces acting on each particle.

(ii) Computing the new positions and velocities, by integration of the equations of motion, and

(iii) Evaluating physical properties.

Step i) is the most consuming computer time, involving a calculation of order n^2, being n the number of particles.

In this simple example a serial MD program could look like this:

```
do i = 1, n-1
    do j = i+1, n
        Compute the interaction of the
         pair (i,j)
         f(i) = f(i) + interaction(i,j)
         f(j) = f(j) - interaction (i,j)
    end do
end do
```

Before considering how one should try to parallelize this two nested loops, let Craft try it in a naive way, just distributing the first loop between the available processors. The speed-up obtained is shown in the lower curve of Figure 3.

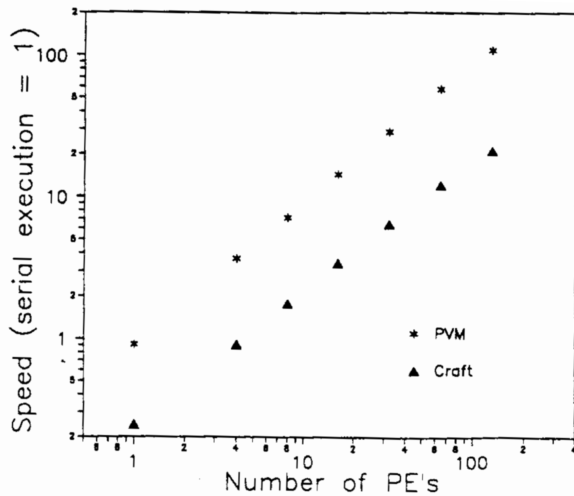

Figure 3

It is clear that although the speed-up scaling is suprising, the starting point (the execution in a single PE) and thus all the curve is not adequate.

One delightful solution to this problem consists in distributing the data (positions and forces) relative to the n particles in $2\ N_{PE's} + 1$ groups (see Figure 4):

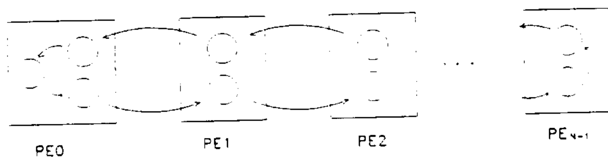

Figure 4

A complete calculation of all the pair forces over all particles is achieved after $2\ N_{PE's} + 1$ steps, each one consisting in the *local* computation and addition by every processor of the interactions between two groups of particles (in the case of the PE0 only the interactions between the two rightmost groups are computed), followed by a subsequent communication of positions and forces. This procedure fits precisely in a message passing code with correct speed-up, since the two relevant issues are satisfied: all computations are local, and the need of communication, although rather particular, is not excessive. In fact, in Figure 3 (upper PVM curve) the efficiency of this procedure is shown to be much more satisfactory than the previous Craft one. The question is therefore whether the Craft model programming is flexible enough to allow a procedure of this type to fit naturally and easily into it.

The answer to this question is that it is equally easy to describe (and execute efficiently) this type of algorithm in Craft than in PVM or any other explicit message passing tool. A simple way to accomplish this is setting a parallelism between an explicit message passing code and a "subset" of Craft. In fact, all variables and arrays in PVM are local (or private), and they have an extra index not quoted explicitly (its processor number) that plays a decisive part in the algorithm. An identical foothold is achieved in Craft if one uses the same problem-decomposing variables and arrays, declaring all index degenerate, except for and additional one (the implicit processor number in PVM-like codes) declared as distributed. Then the two major characteristics of explicit message passing are held:

- All computations are local: a shared do loop with the loop index standing for the PE number aligned with an array of the type just described is known to be local by the compiler.
- Any communication required is specified by the user. When needed, the explicit communication is specified by en instruction like

$$x(:,: \ldots, idest) = x(:,:, \ldots, isource)$$

and this "subset" of Craft may be thought in terms of a particular explicit message passing model, in which the syntax has been chosen to closely follow the fortran notation.

Following this approach, a second Craft version for this problem has given rise to the much more satisfactory speed-up curve shown in Figure 5.

It is clear that the Craft 2 version closely follows the ideal scaling, contrary to the naive preliminary version. Any differences in performance between the Craft 2 version and a code

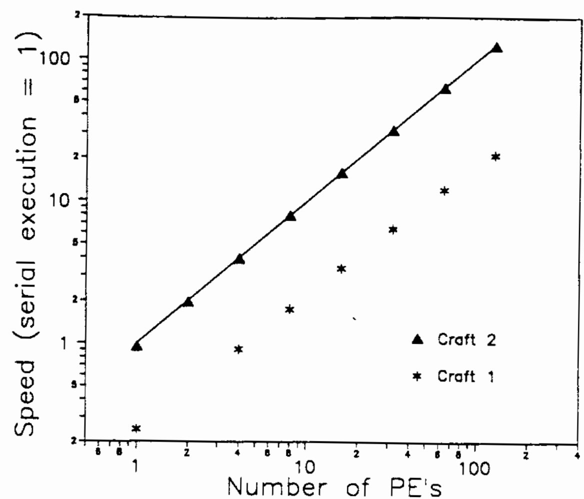

Figure 5

using remote memory get/put calls or explicit message passing is now due only to the particular implementation of the mechanism responsible for the data transport. In the main point, the correct description of the parallel algorithm, it is clear that Craft is perfectly able to achieve its part. Furthermore, it closely resembles a traditional get/put or message passing program, with the additional advantage that it can be debugged, tested or run in any conventional computer.

A supplementary amusing question arises: despite the fact that Craft is (or may be put) on the same foot as a PVM-like program, is there some feature of a message passing programming model that Craft cannot do? The answer is that nonblocking send or receive calls are outside the Craft possibilities. However, this is not a fundamental deficiency, and the use of parallel constructors would allow that feature. Parallel constructors are used for example in Occam language, and run in Transputer systems. They are most useful in allowing the simultaneous use of the "computing" and "communicating" hardware, and allow the user to exploit nonblocking remote reads or writes in a very clean and comfortable style. Its translation to the Craft directives could be something like this:

```
cdir$   parallel(1)
        Essentially communication code
cdir$   end parallel(1)
c
cdir$   parallel(1)
        Essentially computation code on a
        different set of variables
cdir$   end parallel(1)
cdir$   barrier
```

the two parallel sections tied by a common number (1 in the example) meaning that these two parts of code can be executed at the same time. In principle, such a piece of code could be time saving irrespective of whether placed in a parallel or sequential region of the program.

Some considerations may be derived from what has been exposed:

1. Craft can be viewed as having a limit in which *it is* an explicit message passing programming model, with all its full power for describing parallel problems.
2. Releasing the tight conditions that lead to this limiting case, a scope of possibilities arises. In this sense, small changes in a serial source code allow achieving high efficiencies in intermediate problems.
3. Efficient and general Craft programs can be written in such a way that they can be equally executed in serial machines. This may be very important when debugging or adding and testing new pieces of code.
4. It clearly detaches the task of describing the parallelism of the problem (including the suitable communication pattern) from the task of implementing the communication calls. The first one, which the compiler cannot find efficiently by itself is left to the user, while the second one can be afforded and fully granted by the compiler or the underlying hardware. A Craft code would have then the promise that future improvements in transferring data will be automatically used in a totally user transparent fashion.
5. Non-specialist parallel programming people can collaborate efficiently with a more experienced user. It permits to take benefit of parallel architectures to people that otherwise would not consider this possibility.

In summary, for simple parallel problems, where more complex approaches would be unnecessary, Craft provides and easy and quick tool to obtain a running parallel program with good efficiency. On the other hand, for more difficult problems from the point of view of parallelism, the Craft system can be used as a particular message passing or get/put model programming, using a fortran familiar syntax, achieving then similar performances to more advanced systems. Furthermore, in both cases it makes possible the desirable attribute of keeping compatibility with common serial machines. It thus meets the requirements for allowing a "continuous" transition from sequential to parallel programming, with the warranty of keeping good efficiencies in general parallel algorithms.