An Introduction to the APE100 Computer

N. Cabibbo
INFN, Sezione di Roma II and Dipartimento di Fisica, Universita' di Roma Tor Vergata

F. Rapuano
INFN, Sezione di Roma and Dipartimento di Fisica Universita' di Roma La Sapienza

R. Tripiccione
INFN, Sezione di Pisa, and Dipartimento di Fisica, Universita di Pisa

Abstract

This paper describes the physical background that motivated the development of the APE100 parallel computer for theoretical physics. The architecture of this class of computers is then described and some examples are given of the programming style used on APE100.

1 Overview

In the last few years many projects of specialised computers for theoretical physics calculations have started. Most developments have concentrated on the design of specialised processors for Lattice Gauge Theory (LGT). LGT's are a framework to perform first principle calculations in fundamental interactions which on the one hand can be easily simulated numerically but require a huge amount of computer power to handle realistic problems.

In this paper we start by presenting a few simple ideas on the basic features of the physical problem, in order to give a feeling of the enormous computer resources required in this field. We then describe the very simple architecture of the APE100 parallel processors that has been directly shaped by the computational requirements of LGT. Section four is devoted to the software architecture of the APE100 computer, and is followed by some concluding remarks.

This paper covers only some basic aspects of the architecture and the implementation of the machine, and tries to give a feeling on its use by means of some simple examples. The interested reader is referred to [1] for more details.

2 The physical problem

Today we have reached a deep knowledge of the interactions that we observe in nature. There are four well known interactions: Strong, Electromagnetic, Weak and Gravitational, respectively responsible for the cohesion of protons and neutrons in nuclei, the interaction among electrically charged particles, ß decays and the attraction force among massive objects.

For the first three, starting from first principles of symmetry, we have a unified theory that gives a description of these natural phenomena in very good agreement with the experiment and respects the requirements of special relativity and quantum mechanics. The unification with other types of interactions has not yet been achieved for Gravity.

Having a theory is not enough to predict experimental results, one must be able to actually perform the calculations. This is in general a very difficult and largely unsolved problem for quantum field theories such as those which describe the fundamental interactions. There is a well established procedure which can be applied when the coupling constant (i.e. the strength of the force) is small. In these cases we can apply perturbation theory, which consists in imagining the system under study as a system without interactions modified by a small perturbation due to

the interaction itself. In this way we can express physical quantities as a power series in terms of the small coupling constant. Perturbative methods are very successful in treating the Weak and Electromagnetic Interactions, whose coupling constant is small and a Perturbative treatment is possible in all the reasonable ranges of energy involved in High Energy Physics experiments.

For the strong interaction the perturbative method fails in general, with the notable exception of a class of very high energy phenomena, the so called "deep inelastic collisions". In most other cases the perturbation method fails miserably. In these cases on can try the method of numerical simulation of the theory. Numerical simulation of quantized theories is a very interesting method, since it is conceptually very simple, and in principle capable of yielding results of arbitrary precision. The precision is in practice limited by the available computer power. This happens for two different reasons. The first is that in a field theory the system has an infinite number of degrees of freedom, corresponding to the value of the fields at all points of space and time. In order to simulate the behaviour of such a system on a computer on must start by representing continuous space and time by a lattice of points. This does not by itself limit the precision, since one can always improve it by increasing the number of points. In doing so one is however limited by the available computer power. Current simulation of Quantum Chromo Dynamics (QCD) - the theory of strong interactions use a lattice of 32^4 points, 32 along each direction for a total of about one million points. Increasing the number of points by a factor 2 along each direction would increase the total number to about 16 million. It is clear that the limit is soon reached even with the most powerful machines now available.

The second source of imprecision arises from the method of simulation, which is a statistical method very close to the Monte Carlo method used in the simulation of high energy experiments. The essential aspect which distinguishes a quantized system from a classical one is the presence of fluctuations. If we denote by $\{X_i\}$ the ensemble of variables in the problem, i.e. the fields at all points in the lattice, the theory does not specify a single value of $\{X_i\}$, but a distribution of possible values. We note that the lattice points represent both space and time coordinates, so that $\{X_i\}$ corresponds to what we would call a trajectory for the system under study, so that the quantum theory specifies a distribution of possible trajectories. The expected value of a given physical quantity, $O(\{X_i\})$ is a suitably weighted average:

$$\langle O \rangle = \sum \exp^{-A(\{X_i\})} O(\{X_i\}) \qquad (2.2)$$

In techical terms the weight of each trajectory is the exponential of the corresponding "action" changed in sign. To evaluate expressions of this type one has to use the Monte Carlo method, which consists in producing a possibly large number N of configurations (trajectories) chosen with the correct probability among the infinite number of possible ones. One then writes:

$$\langle O \rangle = (1/N) \sum_{k=1}^{k=N} O_k \qquad (2.5)$$

where O_k is the value of O on the k-th configuration. A statistical error is intrinsic to this procedure which decreases as $1/\sqrt{N}$.

In conclusion the precision of the simulation increases with the size of the lattice and the number of trajectories which are used. Both avenues for improvement imply large increases of computer resources.

Numerical simulation of quantum field theories is easily implemented on parallel computers, since the algorithms used for generating the configurations are essentially local, which permits the computation of the new values of a set of points to be carried on in parallel. It is

straightforward to partition the mesh of points into blocks that can be handled independently by different processors. Each processor must perform the same set of operations.

The processors are not fully independent, however, since some communication capabilities are needed to deal the cases when the neighbours of a point assigned to a given processor belong to different processors.

The computation needed in the simulation of LGT is local, since the interactions in a field theory only involve the fields at neighbouring points, and homogeneous, since the fields obey the same laws at all points. These properties are fully exploited in the specialised processors that have been built for LGT's. In fact, since all nodes perform the same stream of computation, they can share the same control structure. Also, the interconnection network among the nodes does not need to be too sophisticated. Substantial savings are therefore possible in the complexity of these machines, and machines much more powerful than traditional computers have been built in this way.

There are other reasons why dedicated LGT engines can easily reach high performances, connected with the nature of the algorithm. First, computational kernels encountered in LGT have a very low ratio of needed data items over floating point operations. This means that a relatively low memory bandwidth is necessary to sustain the performance of a processor. Code-development is also rather simple: a typical LGT program is made of a few thousand lines and contains just three or four main modules nested in an appropriate number of loops (these modules are of course the same for all processing nodes, for all homogeneous applications) It is therefore not unreasonable to rewrite the whole program from scratch, to adapt it to a new type of computer.

The computation strategy described above is straight-forward in principle. In practice, the extraction of physically significant results has required an exponentially increasing amount of computing power. This is shown in fig. (1) (adapted from [3]), where the number of floating point operations that have been used typical LGT studies is shown as a function of time. The numbers of fig. (1) span a wide range of values growing exponentially at a rate of a factor ten every two years, from a few VAX CPU hours in the late seventies to thousands of CRAY level CPU hours in the middle eighties. These very large requirements explain why "theoretical' physicists have decided (in the early eighties) to build their own LGT computers.

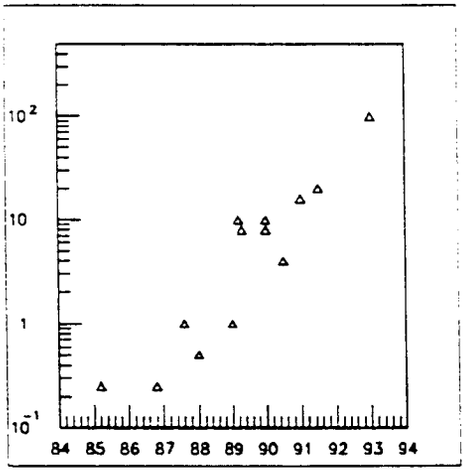

Figure 1: Number of floating point operations performed in typical investigations in LGT, plotted versus time (adapted from [3]).

Fig (2) shows the peak power of several specialised computers for LGT as a function of the year in which they have been completed. Two features are worth mentioning in Fig. (2), namely the growth-rate, almost equal to that of fig. (1), and the peak speed of the most recent such machines, already in the 100 GFlops range.

In the following section we shall describe in some details one such engine, APE100. A survey of the evolution of specialised machines for LGT can be found in [4].

3 The APE100 Architecture

APE100 has been designed to perform efficiently on LGT simulations. Some basic considerations have been taken into account when designing its basic architecture:

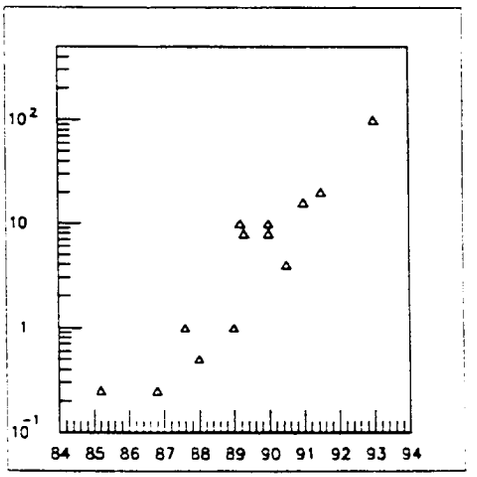

Figure 2: Peak performance (measured in GFlops) of several specialised computers for LGT, plotted versus time.

- APE100 contains a very large number of processing elements, and each processing element has its own local memory. The number of processing elements can be extended at will, in principle. They are arranged in a very simple topological structure, namely a three-dimensional grid with periodic boundary conditions.

-a very simple control strategy for each processing element has been chosen. Just one controller steers the work of all nodes. Each node performs the same operation at each time step, on different data items. This technique is usually called Single Instruction Multiple Data (SIMD) processing.

-It must be possible to program APE100 with a high level language. It must be clearly understood that the partition of the data structures onto the processing nodes and the parallelization of the programs rest completely in the hands of the APE100 programmer. However, once an algorithm has been explicitly parallelized, tools must be available to write the program quickly and efficiently with some programming language similar to standard languages, such as FORTRAN or C.

The APE100 architecture [1] can be summarised very briefly (See fig. (3)). Each node in APE100 has a processor and a memory bank and is able to exchange data with its six nearest neighbours. All nodes execute the same program, steered by just one controller. Nodes specialise in floating point arithmetic, while the controller handles all integer arithmetic of the programs. These are typically limited to address calculations and book-keeping of do-loop indexes. The processing node is the single most important component of the whole machine,

which contains from eight to more than 2000 nodes. It includes a floating point multiplier and a floating point ALU. All operands and results are contained in a large register file, with 128 registers. Such a large register file plays a key role for an efficient execution of LGT programs. In fact, it is needed to store all intermediate results of a long computation. This is important to sustain high performance in spite of limited memory bandwidth.

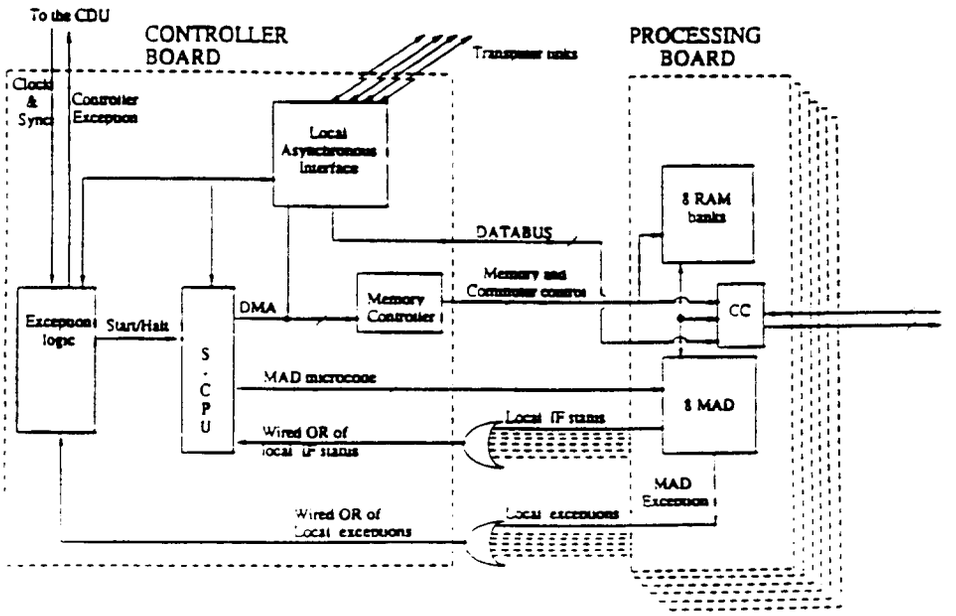

Figure 3: Basic architecture of the APE100 computer.

The architecture of the processing node has been largely shaped by the requirement that the entire node fits into one single integrated circuit, along with all its support circuitry. This integrated circuit - the MAD processor [5] - has a peak performance of about 50 MFlops, so an array of 2000 nodes reaches the target speed of 100 GFlops.

APE100 is connected to a host system (a VAX/VMS or UNIX computer). The host computer runs the cross-compiler for APE100 and interfaces to APE100 to run its programs and to collect results.

At present four 128-nodes, 6 GFlops unit have been built, as well as some smaller software-development machines. Larger units (25 GFlops and 100 GFlops) are expected in 1993.

4 Programming the APE100 computer

A large effort has been done in APE100 to make programming as user-friendly as possible. A new programming language has been developed for APE100. Its main features can be summarised as follows (and will become clear in the examples below):

(1) new types and operators can be defined.

(2) operators can be over-loaded

(3) strong type checking is present

(4) loop expansion at compilation time can be performed and conditional compilation is possible.

This high level language (usually referred to as *APESE*) is based on a so called dynamical grammar [6], able to modify its syntactical rules. We will not discuss its formal properties here. Rather, we will try to present some of its features from the point of view of the average user. The simplest approach to APESE is that of using its flexibility in order to cast its structure so that it looks very similar to standard FORTRAN or C. This has already been done and saved in a personalization library, that is loaded every time the compiler is invoked. This feature is of course particularly useful for a smooth transition towards APESE of start-up users. A more innovative approach, on the other hand, makes full use of the new features, in order to make programming simpler, quicker and self-documented. As an example, let us define a 3D vector as a new type, and the external product as a new operation on this newly defined type. This can be easily done in two steps: First, a new type is readily defined to be a collection of three real numbers:

```
record 3d_vect
   real x,y,z
end record
```

Next, new operations are defined on the newly created type, using some (hopefully easily understood) basic instructions of the APESE language. The external product, for example, could be defined in this way:

```
3d_vect -> 3d_vect^a "*" 3d_vect^b
{
temporary 3d_vect res
res.x = a.y*b.z - a.z*b.y
res.y = a.z*b.x - a.x*b.z
res.z = a.x*b.y - a.y*b.x
return res
}
```

The newly defined types and operations can now be freely used. Consider for example this fragment of code.

```
3d_vect  A,B,C,D
real v
C = v*(A*B+D)
end
```

Note in this example the different meaning of the * sign in the r.h.s. of the equation for C. A*B implies an external product, while v * (...) is a scalar multiplication of each component of the 3-D vector in parentheses.

The same set of instructions in a standard (FORTRAN-like) language would look like the following lines:

```
real  A(3),B(3),C(3),D(3),TMP(3)
call EXT_PROD(TMP,A,B)
Do i = 1,3
C(i) = v*(TMP(i) + D(i))
End DO
```

The first version is clearly more easily readable, does not require use defined temporary variables, and is automatically expanded in line (this has some consequences in the optimisation process, see later).

The APESE language is a compromise between two main classes of languages, that is, special purpose language, which are extremely efficient but able to handle a limited class of problems, and general purpose languages (such as assembly level languages) that can cope with every class of problems and can be very well optimised but need long and careful programming and result in clumsy and difficult to maintain codes. In the case of the APESE language, new types and operators can be used to specialise the code to a particular problem, code writing becomes simpler but performance is not usually decreased (see later).

As already stressed, a big effort has been made to optimise the executable code. The optimisation process goes through several steps that are taken after the compiler has produced an intermediate assembly level code:

- all unnecessary memory moves are removed.

- all operations whose result is known at compile time are removed.

- *dead branches*, (i.e. instructions whose results are not used) are removed. A typical example are the instructions to compute non-diagonal elements of a matrix during the computation of its trace.

- operations are re-organized in such a way to cast them in the so-called normal form. A normal operation is the operation A*B+C. Consider as an example the operation

A1*B1 + A2*B2

where A1, A2, B1 and B2 are complex numbers. The real part of the result is expanded as:

(A1r*B1r-A1i*B1i)+(A2r*B2r-A2i*B2i)

and can be re-arranged in the following way:

(((A1r*B1r-A1i*B1i)+A2r*B2r)-A2i*B2i).

The latter structure uses the normal operation in all places except one and is more efficient because the APE100 hardware can start a new normal at each clock cycle.

At this point, instructions are re-scheduled in order to fill as many pipeline steps as possible. Two obvious constraints must be respected in this process:

- the dependency graph must be respected (that is, an instruction cannot be started if its arguments have not been evaluated).

- two instructions cannot use the same hardware device at the same time.

Very good performance levels are obtained in this way: kernels of LGT codes have reached performances larger than 70% of the peak value.

5 Conclusions

We believe that APE100 is now a valuable facility for intensive theoretical calculations. The effort spent in the design and development of this machine has certainly been worth while, as now thousand of APE CPU hours are available to theoretical physicists, who can tackle problems that would not have been possible with standard commercial computers only.

We would like to thank all the organizers for the very pleasant and interesting School that they have set up in L' Aquila.

References

[1] A. Bartoloni et al., "The APE100 Computer: I, the architecture", *International Journal of High Speed Computing*, in press.

[2] K. Wilson, *Phys. Rev.* D14, 2455 (1974).

[3] N. H. Christ, "QCD Machines, Present and Future", *Proceedings of Computing for High Luminosity and High Intensity Facilities*, pp. 541-548, Santa Fe, NM (April 1990).

[4] D. Weingarten, *Nucl. Physs.* B (Proc. Suppl.) 26, 126 (1992).

[5] A. Bartoloni et al., *Particle World* **2**,65 (1991).

[6] S. Cabasino, P. S. Paolucci and G. M. Todesco, "Dynamic Parsers and Evolving Grammars", *ACM SIGPLAN Notices* 27 (1992).