INSTITUTE FOR FUSION STUDIES

DOE/ET-53088-483

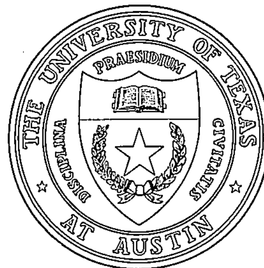
IFSR #483

IFS Numerical Laboratory Tokamak

M.J. LEBRUN and T. TAJIMA
Institute for Fusion Studies
The University of Texas at Austin
Austin, Texas 78712

March 1991

THE UNIVERSITY OF TEXAS



AUSTIN

IFS NUMERICAL LABORATORY TOKAMAK

M.J. LeBrun and T. Tajima

Institute for Fusion Studies
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

A numerical laboratory of a tokamak plasma is being developed. This consists of the backbone (the overall manager in terms of the MPPL programming language), and the modularized components that can be plugged in or out for a particular run and their hierarchical arrangement. The components include various metrics for overall geometry various dynamics, field calculations, and diagnoses.

I. INTRODUCTION

Large-scale simulation has been very successful as a design and diagnostic tool in a wide variety of applications (integrated circuits, jet aircraft, nuclear reactors, to name a few). Similar application of large codes to gain a predictive capability in magnetic confinement may be possible, in light of algorithmic improvements, improved software developing tools, and faster computers. We earlier suggested such an approach.[1] The success of this effort depends critically on the choice of programming language, source file management, execution environment, and coordination of improvements to the code.

This "numerical laboratory" one wishes to construct for plasma confinement and transport studies must contend with many obstacles and goals:

a) The system as a whole is very complex.

b) Different levels of physics interests exist such as time scales, spatial scales, microphysics vs macrophysics, etc.

c) Many different (and possibly newer) algorithms have emerged and will do so in the future. Our backbone should be able to handle these flexibly.

d) Our code should be adaptable to anticipated radical changes in computer architecture. Some leading contenders at present are massively parallel machines and desktop supercomputers.

We highlight our tokamak numerical laboratory at the IFS in the following way: (i) the overall manager (programming language), (ii) general and flexible metric, (iii) δf algorithm, (iv) δf implementation of gyrokinetics, (v) various different reduced or full dynamics, (vi) various local/global boundary conditions, (vii) electrostatic/electromagnetic interactions, (viii) explicit and implicit advancements, and (ix) system-independent input-output and graphics. In particular, we focus the item (i) in this short communication.

The ideas expressed here are based on an existing code, TPC (Toroidal Particle Code)[2]. TPC is roughly 50,000 lines, written in MPPL (a preprocessor to Fortran), and runs on several platforms (CTSS, Unix, VMS, UNICOS port in progress).

(paper presented at the U.S.-Japan Workshop on Advances in Simulation Techniques Applied to Plasmas and Fusion, on Sept. 27, 1990.)

II. THE OVERALL MANAGER: PROGRAMMING LANGUAGE

Why have we chosen MPPL? MPPL, along with its "big brother" BASIS, was designed to repair many of the deficiencies of Fortran as a tool for writing and maintaining large codes. MPPL has improved flow control modeled after the RatFor preprocessor and a subset of the macros in the M4 macro processor. These are combined together in a more robust and elegant way than is possible if one uses RatFor and M4 in sequence, and is more powerful than either one used separately.

Some features of MPPL are:

- Improved flow control, in the spirit of Fortran90 constructs, which allows modularity and hierarchical code construct.
- A formal macro language, including conditional compilation.
- Relatively portable -- runs under CTSS, LTSS, Unix, VMS, UNICOS.
- Minimal learning curve since based on Fortran, which most scientists already know.
- Generates ANSI standard Fortran77 output.

Disadvantages of MPPL:

- Major inadequacy of Fortran 77 still present -- lack of true data structures. External data structures are still possible with macros, however.
- No source-level debugging (need to refer to generated Fortran source file). Relatively minor inconvenience.
- Currently owned by University of California and not freely distributable (this will change).

Some alternatives to MPPL, and comments:

- Fortran77. Unacceptable since it is not powerful enough.
- Fortran77 and Precomp. Also unacceptable since Precomp is not portable. Further, MPPL is much more powerful.
- Fortran77 and CPP. Very portable, but limited.
- Fortran90. It is not yet available.
- Ratfor/M4. Might as well use MPPL, although RatFor and M4 are more widely available. Problems encountered with this approach led to creation of MPPL.
- C. A reasonable choice, however, in some ways C is not as well suited as Fortran for scientific computing (multi-dimensional arrays, complex data types). Can be harder to optimize. Combining C with Fortran codes is nonportable and can be difficult. Learning curve can be steep for scientists used to working in Fortran.
- C++. More suitable for scientific programming than C, but other issues remain. Optimization is a bigger question mark. Compilers may still not be widely available (no C++ compiler at NERSC). Learning curve is steeper yet.
- BASIS, which is based on MPPL. A good choice, but this limits the code portability since BASIS does not run on all available platforms at present (this could change).

III. MODULARITY AND HIERARCHICAL CODE CONSTRUCTION

Use of improved flow control (over standard Fortran) results in more clear, robust, and modular code. A simple example:

```
do i=1,NX
   if( a(i) < eps ) then
      warn('a(i) too small, continuing..')
      break
   endif
   b(i) = 1. / a(i)
 enddo i
```

Here the break terminates the loop, after a warning message is printed ('warn' is a macro). MPPL has constructs 'break' for exiting a loop (similar to 'exit' in Fortran 90), and 'next' for proceeding to the next iteration (similar to 'cycle' in Fortran 90). Because of the absence of labels, the function of this loop is clear; more importantly, it can be used *as-is* anywhere in the code, encouraging modularity. Constructs 'while', and 'case' are also available.

Macro data structures encourage modularity by storing only *one* copy of data declarations which is expanded by a routine needing access (similar to that of cliche in Precomp). It is typically most productive to store all macro data structure definitions in separate files, included into only those source files which reference them (using MPPL 'include' statement).

Hierarchical code construction ensures that the code is "reusable", i.e., resembles a set of tools that can be trained to some given task which depends on the need of the user. This can include different dynamics types, multiple particle or fluid species, and so on. The main program is best cast in the form of a generic initial value problem.

Conditional compilation extends the power of the code in many ways. One of the most important is portability. In the code example, we hide system-specific startup and exit code in the 'code-startup' and 'code exit' macros. MPPL's built-in SYSTEM macro is sufficient to handle most incompatibilities between systems. Where this is not true, one can define macros for machine, site, and compiler. Situations where conditional compilation is useful include:

- System-dependent i/o
- System-dependent code profiling facilities
- System-dependent compiler enhancements
- Other system calls, such as dynamic memory allocation

Conditional compilation can extend power of code in other ways as well.
It facilitates compile-time selection of:

- Dynamics types
- Field equations and/or field solver
- Graphics
- User-specific code enhancements

Macros can be used to substantially enhance the power of the language. Some of the capabilities of the macro package employed by TPC include:

- Enhanced output primitives
- A portable code timing facility
- Error diagnostics
- Generic compiler directive macros

This package will be made publicly accessible to MPPL users at NERSC in the near future.

IV. SYSTEM-INDEPENDENT INPUT-OUTPUT AND GRAPHICS

System-independent input-output are helped by adoption of low-level primitives which achieve the desired goal for each system. These are used uniformly throughout code. In TPC the i/o is implemented via the low-level routines.

System dependencies are hidden inside these routines. The user sees no difference in usage regardless of the platform. These have call syntaxes similar to the NERSC fortlib routines of similar name.

Inter-machine portability of binary files has always been a problem, but this can be solved via adoption of a common binary file format such as HDF (hierarchical data format), developed at NCSA. This allows one to write data output files on a Cray, for example, and transport them to a workstation for post-processing analysis.

REFERENCES

1. T. Tajima, *Computational Plasma Physics*, (Addison-Wesley, Redwood City, 1989) 'Epilogue'.

2. M.J. LeBrun and T. Tajima, IFSR #374 (1989).

This work supported by the U.S. Department of Energy