Tools to facilitate the implementation of grid based finite difference algorithms on distributed computer systems.

David W. Jones☮, John P. Hulskamp☮ and A/Prof. David Abramson*,

☮ Centre for Concurrent Computing, Dept. Computer Systems Engineering, Royal Melbourne Institute of Technology, Melbourne, Victoria, Australia.

* School of Computing and Information Technology, Griffith University, Nathan, Queensland, Australia.

email: davejones@rmit.edu.au

Abstract

A set of tools has been developed that eases the implementation of finite difference engineering and scientific applications with distributed computer systems. The tools have been implemented for transputers using the Helios operating system and have been integrated with the MicroEmacs editor to simplify their use. The tools consist of a generic main program, a set of routines and a suite of macros. The main program provides all of the coarse grain steps and synchronisms for the implementation of finite difference algorithms on a distributed system. The user need only create and link in functions for initialising, manipulating and communicating the data as well as functions for reporting summative results. There are routines for the scattering and gathering of data as well as for reporting summative results for a system. The suite includes macros that abstract the synchronism and data communication to a conceptually high level, whilst transparently verifying communications.

Keywords: Transputer, Helios, finite differences, distributed systems, software tools.

1. Introduction

Many physical systems are modelled by partial differential equations of two dimensional data structures. These structures can be discretized [1,2] onto a grid and in so doing the equations reduce to finite difference equations. The system is then iteratively solved by applying the difference equations at each point once for each iteration. Boundary value problems require that the iterations continue until the grid values show some measure of

convergence, whilst initial value problems require one iteration for each time step. The finite difference equations exhibit only localised data dependencies and therefore tend to perform efficiently with distributed computing platforms using point to point communication channels.

A finite difference algorithm is usually implemented using a master with subordinate slaves with multiple processor systems. The master process initialises the data structures for the grid, sends each slave their portion of the grid, receives the results back from the slaves and provides a final report for the task. In a distributed system it can also provide synchronism for the slaves. Each slave process performs the operations upon their domain of the grid. Because their sequencing is quite deterministic, it is possible to write a prototypal distributed algorithm for finite difference methods acting upon a grid.

A generic algorithm for such systems has been written which encapsulates the major steps that all such algorithms perform and has been implemented as part of a software tool for an array of transputers using the Helios operating system. This toolset implements the generality as well as providing a proforma for the application specific data manipulations and for their specification. It also provides a suite of macros that abstract many aspects of the algorithms including interprocess communications and synchronisation which makes the code simple to write, easy to maintain and modify, whilst inserting some communication diagnosis to check for errant communication channels. The user need only define the required data structures and implement the application specific steps as the generic algorithm transparently binds them all together. Also, some routines have been included for manipulating arrays of floating point numbers and for simplifying their dissemination and collection to and from the slave processes.

The toolset focuses upon facilitating the implementation of finite difference algorithms using distributed computing systems whilst providing portablity of code so produced between such computing platforms. The toolset has been named **Babbage** after Charles Babbage's mechanical finite difference machine.

2. Portability issues

Ideally one would be able to take a sequential program written for a uniprocessor system and have the parallel constructs inserted by a parallel compiler. Whilst much work is being done in this area for specific parallel systems, in general parallelization requires insertion of machine specific calls into the code and provision for the configuration of the multiple processes. A parallel program written this way is then bound to a particular system. Portable programming tools such as the Argonne Macros [3] and PVM [4], consist of a suite of calls to provide the parallelism. These calls are at sufficiently abstract level so as to not be system specific and they are implemented on a system by writing them in terms of the parallel constructs available on that system. A parallel program that is written using only these abstract calls for parallelism can then be compiled on any system which has an implementation of these calls.

As stated in [3], there is a wide variety of multiprocessor systems available. They can though be classified into three groups: shared memory, distributed processors and clusters. Shared memory architectures provide support for a number of processors using the common memory. Distributed systems provide support for interprocessor communications (message passing) with each processor has its own memory. Clusters are a hybrid of both shared and

distributed systems. Each cluster of processors has its own shared memory with intercluster message passing. Tools produced for multiprocessor systems that provide portability normally focus upon only one of the above groups. This paper is targeted at distributed systems.

As a minimum, a suite of portable processing calls for distributed algorithms needs calls to send and receive data between processes, calls to provide interprocess synchronism and a method to perform task configuration. It is normally a simple matter to implement abstract SEND and RECEIVE calls in terms of system specific message passing functions to provide interprocess data communications. Synchronous message passing as used with transputers, can be used for synchronism. Systems with buffered message passing usually support a SendResponse system call that requires a response from the receiving process. This can also be used for synchronism. Configuration is more difficult to generalise. The configuration of a distributed task can be provided by a resource script external to the code or by calls that create a process on another processor. Abstracting the task configuration as part of a suite to support portability can be quite difficult. It is better to embed the configuration into the implementation or to include this in the form of an "INIT_ENV" call to initialise the parallel distributed environment.

3. Description of the toolset

This toolset provides for simple implementation of physical systems represented as finite difference equations acting upon elements of a rectangular two dimensional grid with equidistant grid points. Grids can be finite or wrap around to form a torus. The tools are specified for distributed computing systems that have localised point to point message passing. The toolset aims to encapsulate those aspects of such implementations that are universal to all such finite difference equations. It also provides high level tools to simplify the implementation of specific applications.

This version of the software has been implemented for an array of transputers running the Helios Parallel Operating System [5]. Coding is written using the C programming language. The toolset is also to be implemented for the Fujitsu AP1000 [6]. The intention is that a finite difference system can be implemented using the software tools on one of these systems and only require recompilation to be run on the other or any other distributed system for which the toolset is implemented. The calls are implemented under Helios as macros.

This software tool, whilst supporting the minimal calls for interprocess synchronism and data communications, aims at providing a very high level of abstraction. As such it incorporates a number of calls that simplify the distribution and collection of data over a grid of processes. There are also calls to obtain summative data where a scalar value is required from a distributed matrix of values and calls to provide task level synchronism such as a Barrier. The for these calls, although primarily intended for transputers, are based upon the C library for the Fujitsu AP1000 [8].

To write an application using the toolset, one need only enter code into a resource script. This code includes definition of grid data structures as well as the application specific implementation of certain functions. A prototype is provided in the generic resource script for these functions and a suite of high level macros is available to simplify their implementation.

The makefile for the toolset drives the compilations of these functions and links them with the code for the generic algorithm as well as implementing the task configuration.

The MicroEmacs [8] editor has been customised for this toolset so that all aspects of the development of an application can be done from within this environment. The macros are available as a menu. Many actions such as file handling and compilation are bound to function keys.

4.Finite difference equations

The time evolution of physical systems set on an XY plane are often described by partial differential equations in space and time or by a coupled set of such equations [1,2]. The equations are often second order in x and y and first or second order in time.

For computational purposes the XY continuum is discretized into a matrix of equidistant points on a rectangular grid. Time is also discretized into small steps. The partial differential equations consequently become finite difference equations, such that the next value in time at a point on the grid is a rational function of grid values near that point. Being at worst second order means that only grid values immediately adjacent to the point are involved in the finite difference expression.

For example the flow of heat on a 2 dimensional surface, with all constants equated to 1 for simplicity is expressed by:

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=\frac{\partial u}{\partial t} \qquad \textbf{Eq 1}$$

On an XY grid of spacing h the partial derivatives can be approximated with an error of $O(h^2)$:

$$\frac{\partial u_{i+1/2\,j}}{\partial x}\approx\frac{u_{i+1\,j}-u_{ij}}{h} \quad \text{and} \quad \frac{\partial u_{i-1/2\,j}}{\partial x}\approx\frac{u_{i-1\,j}-u_{ij}}{h}$$

which averages to

$$\frac{\partial u_{ij}}{\partial x}=\frac{u_{i+1\,j}-u_{i-1\,j}}{2h} \qquad \textbf{Eq 2.1}$$

Similarly

$$\frac{\partial u_{ij}}{\partial y}=\frac{u_{ij+1}-u_{ij-1}}{2h} \qquad \textbf{Eq 2.2}$$

The second order derivative in x can be approximated by the difference of the two approximate expressions above for the first order derivative in x:

$$\frac{\partial^2 u_{ij}}{\partial x^2}\approx\frac{u_{i+1\,j}-u_{ij}}{h^2}-\frac{u_{ij}-u_{i-1\,j}}{h^2}$$

This therefore becomes

$$\frac{\partial^2 u_{ij}}{\partial x^2}=\frac{u_{i+1\,j}-2u_{ij}+u_{i-1\,j}}{h^2} \qquad \textbf{Eq 2.3}$$

Similarly

$$\frac{\partial^2 u_{ij}}{\partial y^2} = \frac{u_{ij+1} - 2u_{ij} + u_{ij-1}}{h^2} \qquad \textbf{Eq 2.4}$$

Combining 2.3 and 2.4 equation 1 becomes:

$$\frac{u_{i+1j} + u_{ij+1} - 4u_{ij} + u_{i-1j} + u_{ij-1}}{h^2} = \frac{\partial u_{ij}}{\partial t}$$

Rearranging this for $\mathbf{u_{ij}}$ the equation becomes:

$$u_{ij}^{..} = \frac{u_{i+1j} + u_{ij+1} + u_{i-1j} + u_{ij-1} - h^2 \frac{\partial u_{ij}}{\partial t}}{4} \qquad \textbf{Eq 3}$$

This can then be used to integrate the system over time by iteratively generating new values at each grid point. The next value at a point can therefore be determined from the nearest grid points and the time derivative at that point. The time derivative can be determined from the difference between successive values at a grid point or alternatively if there is an identifiable flow in the system with a velocity, then the time derivative can also be expressed in terms of grid differences.

5. The distributed algorithm

The task is distributed between a master process and multiple slaves as in figure 1. The master sets up the grid, communicates it to the slaves, gets their results back as well as synchronising the slaves. The slaves receive their portion of the grid from the master and perform the difference equation on their domain. They then report the required results to the master. The slaves' portion of the grid is determined by geometrically dividing the grid according to the topology of the slaves' configuration. This software configures the slaves as a torus and so the grid is divided up vertically and horizontally.

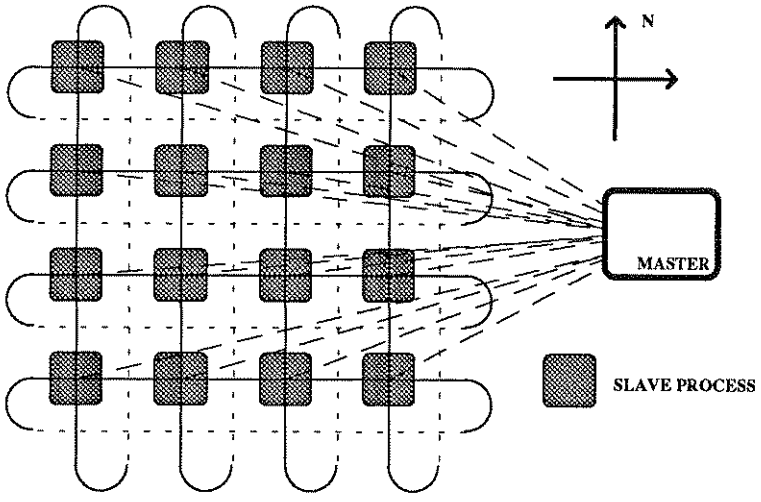

Fig 1. Master-Slaves configuration using a 4 x 4 torus

The steps required to set up, execute and complete a distributed implementation of a finite difference problem are shown in figure 2. The algorithm is shown as a set of coupled processes with the synchronism (double arrow lines) and the message passing (single arrowed lines) providing the binding. Note that there is a single instance of the master process whereas there are multiple instances of the slave process. The application specific steps are shaded. The work routines represent the work to be done for one iteration only.

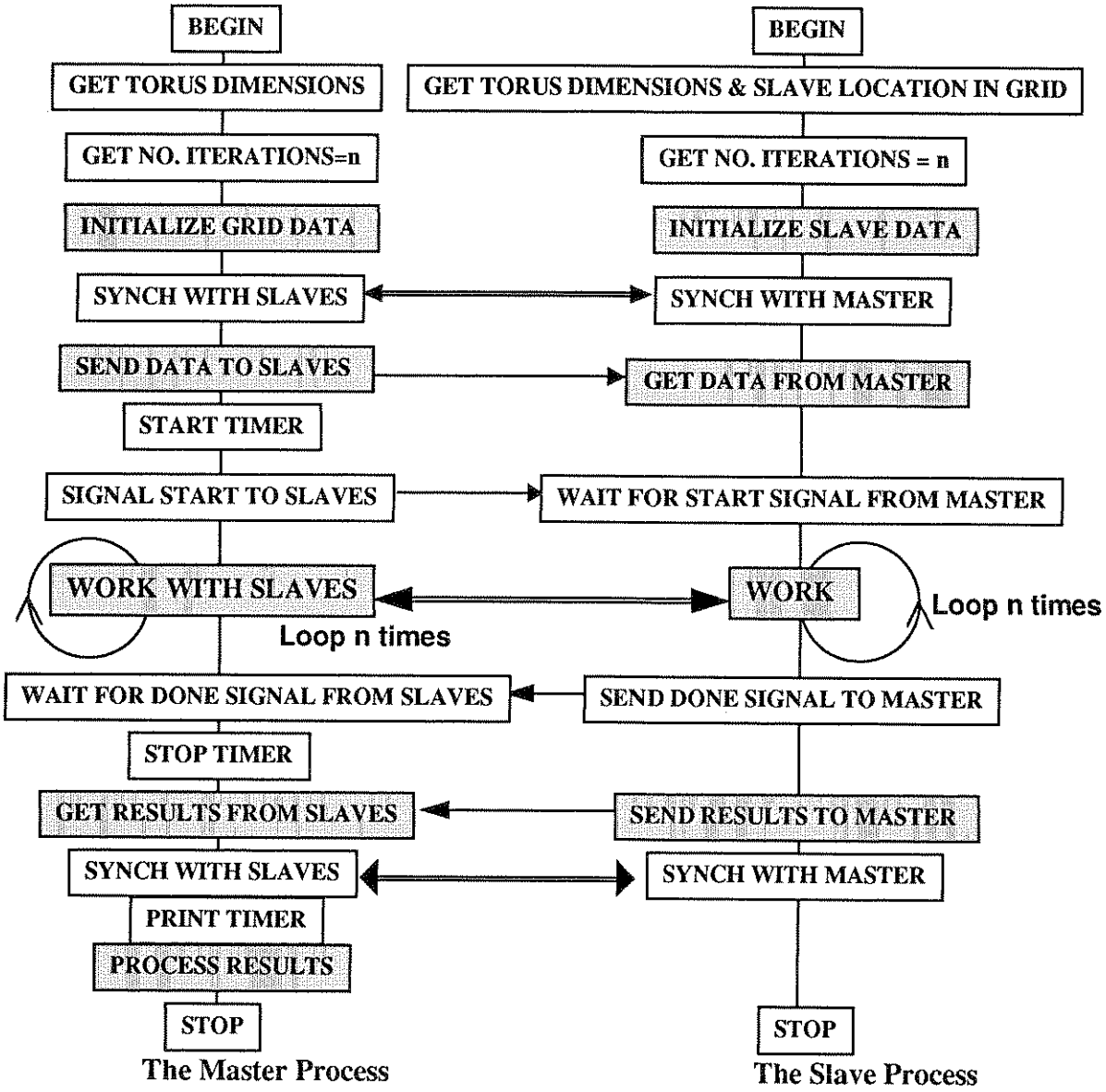

Fig 2. The generalised distributed algorithm for finite difference equations.

6. Target computing platforms

The software has been implemented on the RMIT transputer array and is being targeted at other distributed systems such as the Fujitsu AP1000. Both of these systems have numerical co-processors to enhance computational performance, a large amount of localised memory and efficient point to point communications. Both systems have the processors distributed in a

two dimensional array which wraps around at the edges to form a torus. Hence finite difference applications for two dimensional grids configure simply onto these systems.

6.1. Transputer arrays

The RMIT platform consists of a 4x4 torus of T805 transputers each with 4 Megabytes of memory running at 25 MHz. The torus is connected to 4 other T800 transputers, one of which is the root on a Transtech B004 card mounted in a PC which performs as the host. The root transputer acts as the Helios server for the system running Helios Ethernet 1&2, Helios XWindows and Motif as well as the Helios BSD Fast Filing System. The ethernet and X allows for remote access from an XTerminal. The filing system supports a dedicated hard drive and as such proper Unix file handling is enabled such as extended names, file ownership and symbolic linking.

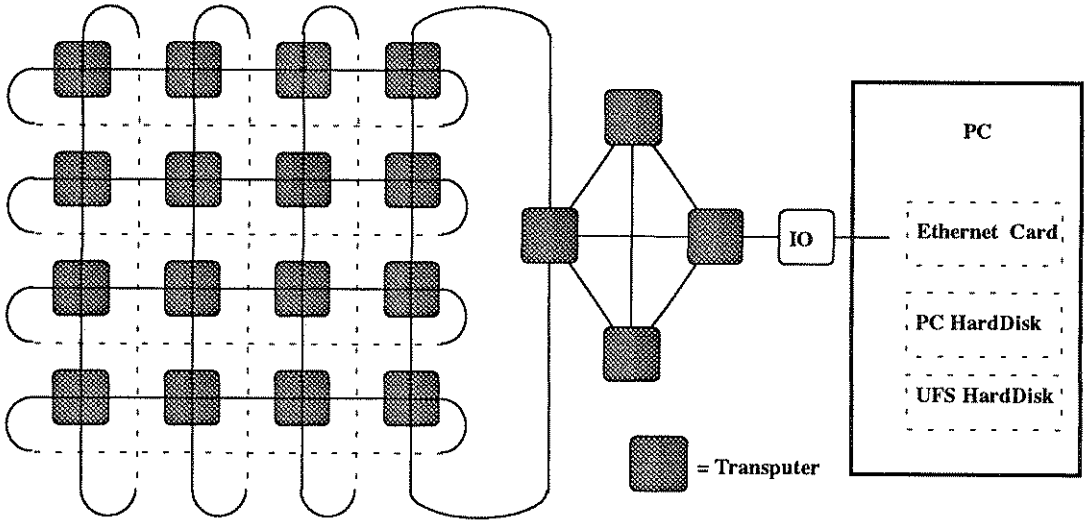

Fig 3. The transputer array

The Helios operating system is a version of Unix that supports transputers in a multiprocessor environment. It has a number of levels of message passing but the recommended one is read and write functions conforming to the POSIX standard because this level guarantees message delivery. A multiprocessor task such as a master-slaves application is produced by writing a separate program for the master and slave process and separately compiling them. Interprocess communications are directly inserted into both source files. The task is configured by writing a script file using the Component Description Language (CDL) [9]. This defines the links for each process type, the numbers of each process type and specifies the interprocess communication channels. This CDL script is at a logical level and does not need to reflect the actual hardware. When the task is run, the processes are placed onto the available processors.

The POSIX read and write calls require a file descripter to specify the required communication channel. This is determined from the syntax of the CDL language. The first level of abstraction makes this easier by introducing macros for direction using names like ToSlave, FromMaster, North and South etc.

The current version of the operating system (V1.3) has high level support for processor farms [10]. This library allows simple implementation of multiprocessor tasks where there is

no need for direct communication between slaves and is therefore suitable for grid applications where there are no data dependencies between grid points when performing computations at each point. The classic example of this situation is the Mandel-Brot algorithm. The farm methodology is not suitable for finite difference equations on a grid because of the inter-grid point data dependencies. This software addresses this issue.

6.2. Fujitsu AP1000

Australian National University has a 128 processor AP1000 machine:

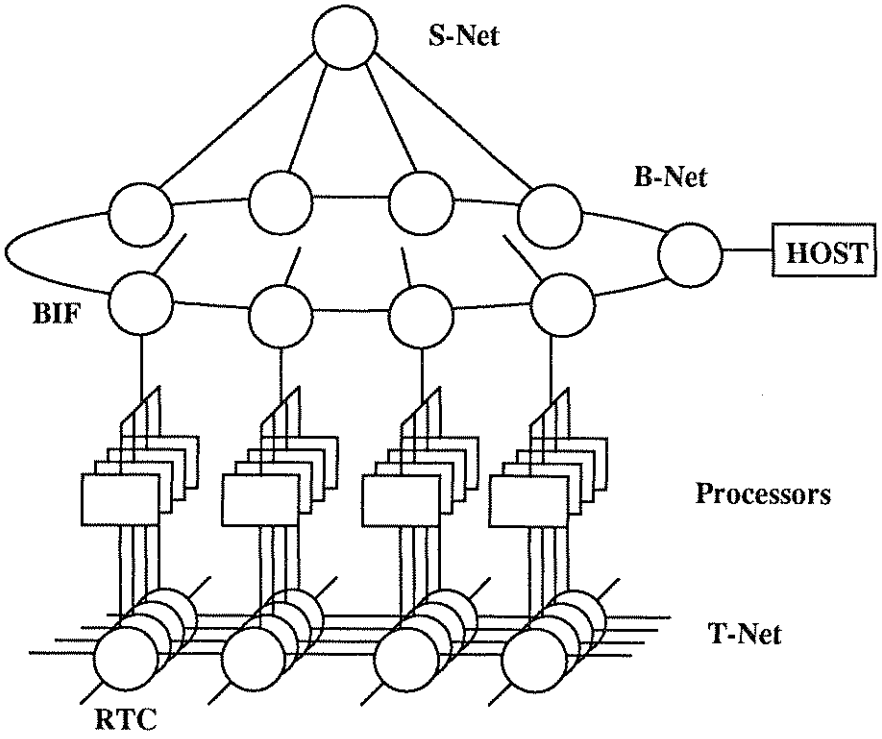

Fig 4. The Fujitsu AP1000

Horie and Hayashi [6] describe the AP1000 as :

> *"a distributed-memory, message-passing, parallel computer. It consists of 16 to 1024 (Sparc) processors connected by three independent networks called the torus network (T-Net), the broadcast network (B-Net), and the synchronism network (S-Net). A message controller (MSC) interfaces processors and networks to minimise message handling overhead."*

A master-slaves application is configured by running the master process on the host and slave processes on the torus. The T-Net uses worm-hole routing and the system calls support nearest neighbour communications as well as any processor to any processor. A separate program is written for the master and slave processes but the master includes system calls to specify the slave process and to configure the torus with the required number of slaves. Communications on the T-Net are buffered and therefore asynchronous as they are handled transparently by the RTC routing chips. Communications can require a response though and

synchronism can be achieved this way. The S-Net is at a signal level and therefore provides for global synchronisation such as barriers and "all cells exited".

7. The toolset contents

7.1. Files

There are six files in the toolset. The first (torus.c) contains all of the code for running a standard application as previously discussed. As such it provides all of the correct sequencing as in figure 2, for the master and slave processes as well as synchronisation in a typical master slave configuration. It is separately compiled to produce both of the master and slave object files. The second file (resource.c) contains all of the functions that are needed to be linked with the master and slave object files to create the master and slave programs. These functions are only shells and therefore require embellishment for particular applications. Not all functions need be added to as the shell functions in resource.c will compile and link as they are.

The third and fourth files are header files that provide differing levels of abstraction of the communication of data between processes. The third file (torus.h) has as its main purpose, the provision of a suite of macros to simplify the aspects of writing communications that are often repeated. These macros put the communications at a high level of abstraction and as such are more readable than the standard POSIX reads and writes. For example, data moves between slave processes by specifying a compass direction whereas POSIX calls require a file descriptor that has to determined from the task configuration.

The fourth file (packet.h) abstracts communications to a very high level such that the system configuration is completely transparent. It contains macros for data scattering from the master and gathering back from the slaves. These automatically share data structures between the slave processes. This file also contains a macro to send edge data between slaves to satisfy data dependencies when performing finite difference calculations. Macros that make use of the Helios V1.3 Vector Library [11] to efficiently determine and report back to the master summative data such as maximum values and averages of the grid elements in slave domains are also part of this fourth file.

The fifth file is a configuration script (torus.cdl) written in the Helios Component Description Language (cdl). Cdl scripts are compiled using the cdl program to create a multiprocessor distributed task. The sixth file is the makefile that drives all aspects of the task compilation including the cdl step.

The only file that is modified for an application is the resource script. The steps that are shaded in figure 2 are implemented in this file.

7.2. Data communication macros

Master:	SCATTER, GATHER, SENDSLAVE, GETSLAVE
Slave:	SCATTER, GATHER, RECVMASTER, SENDMASTER, SENDRECV, SENDEDGES

The SCATTER, GATHER macros require a data structure that is defined for the whole grid. No specifics about domains are required as these are determined by the macros automatically. The other macros require a packet of data which is normally one column of the grid and there are macros for creating and unloading packets. The SENDRECV allows a slave process to send a packet in one direction and receive at the same time from the opposite direction, as is required when edge data is communicated between slaves before or after an iteration. The direction for this macro is specified using NORTH, SOUTH etc when calling the macro. SENDEDGES abstracts this further by providing all of the steps necessary to send all edge data from all slaves to their nearest neighbours.

7.3. Synchronisation macros

Master and Slave:	BARRIER
Master:	SIGNAL_SLAVES, GET_SLAVE_SIGNALS
Slave:	GET_MASTER_SIGNAL, SIGNAL_MASTER

There is a facility for defining messages that can be used in the synchronisms. There are predefined messages such as START, STOP, START_CALC. These are parameters for the synchronisation macros (excepting the Barrier) and so the synchronism is on a specific message. If an error occurs on synchronism it is reported naming the message and the process is aborted.

7.4. Data manipulation macros

Master:	GET_AV, GET_MAX, GET_MIN, GET_AMAX, GET_AMIN, GET_SUM, GET_SUMSQ
Slave:	GET_AV, GET_MAX, GET_MIN, GET_AMAX, GET_AMIN, GET_SUM, GET_SUMSQ

A slave call determines the value for its part of the grid and reports it back to the master. The master collects a result from each slave and determines the overall result.

7.5. Other

Master:	START_TIMER, STOP_TIMER, PRINT_TIMER

8. An example application

As an example, consider the flow of heat on a finite two dimensional square surface. No heat flows into or out of the system but there is an uneven initial heat energy distribution. Using equation one with its simplicity of all constants equated to one, implement the system as a 64x64 grid on a 4x4 torus of processes. Use a grid spacing of 0.1 and a timestep of 0.1. The following would be placed in the resource script:

```
/***** Global *****/
#define h 0.1
#define hsq 0.01
#define dt 0.1
#define XWRAP FALSE
#define YWRAP FALSE
#define NX 64
#define NY NX
float u[NX][NY]; /* The grid data structure */

/***** Master ******/
initData(){
  .../* Initialise u[i][j],the initial heat distribution*/
  ...
}
sendData(){
  SCATTER(u)
}

getData(){
  GATHER(u)
}

/****** Slave ******/
float utimeder[NX][NY],unew[NX][NY];

getData(){
  SCATTER(u)
}

slave(int k){
  int i,j;
/*Initialise utimeder[i][j],the time deriv. of u to 0 */
  if (k==0) {  VgridIJ  utimeder[i][j]=0;  ENDgridIJ }

/* Send edge data to nearest neighbours */
  SENDEDGES(u)

/* Perform finite diff for elements in domain */
  VgridIJ
      unew[i][j]=(u[i+1][j]+u[i-1][j]
        +u[i][j+1]+u[i][j-1]+hsq*utimeder[i][j])/4.0;
  ENDgridIJ

/* Determine time derivative and copy new values to u*/
  VgridIJ
      utimeder[i][j]=(unew[i][j]-u[i][j])/dt;
      u[i][j]= unew[i][j];
  ENDgridIJ
} /* End of slave() */

sendData(){
  GATHER(u)
}
```

Note that the pair of macros `VgridIJ ... ENDgridIJ`, cause iteration over all grid points in a slave's domain. The make file is passed the X and Y dimensions of the processor array and produces a fully configured task as an executable file. This file is then run with the number of iterations given as a command line parameter.

9.Future directions

- The software will be implemented on the Fujistu AP1000 and other distributed computing systems, such as the NCube, Intel Paragon and the Connection Machine.

- As the toolset is integrated with MicroEmacs it is intended that when a synchronism step or message passing is inserted into a master or slave resource script function the user will be automatically prompted for a responding location in the other process.

- A version of the software will be developed so that code will compile and run on a uniprocessor system such as a Sun workstation or an IBM PC. This will allow development of code away from the distributed system and will help catch developmental bugs. For this version many of the macros will be blank.

- XWindows will be used to allow individual processors to output diagnostic messages. General task output will remain the province of the master process.

- Finite difference equations can be expressed as a star diagram [1]. It is intended that a function be written to apply finite differences at a point as expressed as a star diagram. As this function will represent most of the computational time for a finite difference program the function will be written in assembler code to optimise performance.

- The software will be rewritten using an object oriented language. Under this scheme the calls would become class methods. As such the suite will become more extendable, maintainable and robust. The generic algorithm then becomes an abstract class and a specific application is a descendent class of this abstract class.

10. References

[1] Numerical Methods, Software, and Analysis, p311 - 320, J.R. Rice, McGraw Hill
[2] Numerical recipes in C, 2nd Edition, p827-888, W.H. Press et al, Cambridge
[3] Portable Parallel Programs for Parallel Processors, J Boyle et. al., Holt Rinehart and Winston.
[4] PVM 3.0 User's Guige and Reference Manual, A Geist et. al., Oak Ridge National Laboratory publication no. TN 37831-6367
[5] The Helios Parallel Operating System, Perihelion Software Ltd, Prentice Hall
[6] All-to-All Personalised communication on a Wrap-around Mesh, p B-1, T. Horie & K. Hayashi, Proceedings of the Second Fujitsu-ANU CAP Workshop, Australian National University, Nov. 91
[7] CAP-II Library Manual [Host] & CAP-II Library Manual [Cell], Fujitsu Laboratories, 2nd Ed., Jan 90.
[8] The Helios MicroEmacs Guide, Distributed Software Ltd.
[9] The Helios Parallel Operating System, p 157-211, Perihelion Software Ltd, Prentice Hall
[10] The Helios Farm Library, Distributed Software Ltd.
[11] Helios Vector Library, part of the documentation for Helios Parallel Operating System Version 1.3, Distributed Software Ltd.