BABBAGE: A tool to facilitate the implementation of master-slave grid oriented applications on distributed computer systems.

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

☮ High Performance Computer Systems, 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

BABBAGE is a software tool to ease the task of implementing grid based applications that conform to the master-slave model when using distributed computing platforms. Whilst initially developed for finite difference applications, the tool has been extended to support other numerical methodologies for solving partial differential equations. The tool simplifies the coding of distributed grid based algorithms by making the message passing and other configuration issues transparent. The tool is designed for an array of processors and has been implemented on a transputer system running the Helios operating system.

Keywords: Transputer, distributed systems, software tools, grid applications.

1. Introduction

Many engineering-scientific algorithms are discrete implementations of partial differential equations with two or three dimensional grids. With sequential code, a single processor performs the same operation at each grid point. The complete grid is stored within, and operated upon, by the one processor. The operands at each point may be local points as for finite differencing, or they may be global as for spectral methods. When the processing is distributed, the master-slaves model is often used. With this a single processor sets up the grid, distributes it to multiple slaves, and collects the final grid values back from the slaves. The slave processors share the operations upon the grid by each having a designated subregion of the grid to operate upon.

The porting of sequential code onto distributed computing platforms requires the insertion of constructs to provide interprocess message passing and synchronisation. Message passing between the master process and slaves is for the communication of slaves' subregions of the

grid at the start and at the end of computations. The interslave communication is required when the operands at grid points don't reside on the particular processor. Synchronism also adds determinism to parallel programs and can help avoid many pitfalls. Often though it is an overkill because there is inherent synchronism provided by message passing.

The problem with these distributed constructs is that they are machine and compiler specific. If one writes a distributed parallel program for an array of transputers running Helios [1], then that code has to be substantially reworked to run on the same array using ANSI-C [2] and even more reworking is required to port it to the Fujitsu AP1000 cellular array processor [3][4]. There have been attempts to abstract message passing so that calls can be made using the same syntax on a variety of distributed computing platforms. Examples of these are the Argonne Macros [5] and PVM[6]. The former did not find much favour whilst the later currently is in vogue. The problem with abstract message passing is that it still requires a lot of low level analysis of data dependencies and can be very error prone.

This paper discusses the software package BABBAGE that abstracts the master-slaves model and provides routines that move data at a task level. This hides the configuration and synchronism issues and simplifies the implementation of the movement of the grid data between the master and slaves as well as between slaves. Interslave data movement routines have been written for finite differencing as well as spectral methods and other routines can easily be added. The package has been implemented for an array of transputers running the Helios operating system.

2. Portability

The introduction mentions how distributed code can be written at one of three levels of abstraction. With lowest level, parallel constructs are at the machine and compiler level of specificity. Code so produced is not portable. The second level involves using abstract SEND and RECEIVE calls that move packets of data between specific processes. This code is portable between systems that have an implementation of the same suite of message passing constructs but its usually rather difficult to implement in a robust manner. Third method of writing parallel distributed code involves using task level constructs for passing data. With this, one merely specifies the data structure to move when issuing a call to move data. The software tool automatically breaks it into packets and routes the data to the appropriate process. Because the calls at this third level involve no system dependant parameters, the code so produced is highly portable between different distributed systems that implement the software tool.

Examples in BABBAGE of calls at this third level are the Scatter and Gather functions for dissemination of data between the master process and slaves, SendEdges for sending edge data between adjoining slave processes and SendAllRows2Col for exchanging rows of data between slaves in the same column of the grid. Hence the instruction Scatter(pressure,p,press) passes the grid data in the master process called pressure to the grid data structure p in the slaves, tagging it as data type press. The call automatically divides up the master grid between the slaves and so slaves only receive their own subregions of the grid.

By providing a high level of specificity for BABBAGE methods one achieves portability of code, simplicity of implementation and robustness of applications. The code is fully portable between different distributed systems supporting the software tool BABBAGE because the system hardware configuration has been completely hidden from the programmer. The software creates slave processes in a two dimensional grid with the master process having direct access to all slaves. The software tool only needs the ability to map such a logical

process topology onto the real processor topology. This mapping is performed by the tool at compile time without any input from the user except for the dimensions of the slave array. Implementations of grid based applications using BABBAGE are greatly simplified because the data movement methods only require specifications of the data the to be moved. No low level analysis is necessary do such things as create data packets, route them and to place them in the targets' domains. The code is robust because the synchronism required is completely provided for transparently by the software tool. There is no possibility of deadlock nor of errant processes disappearing down a black hole (unless the user puts in infinite loops etc.) Each process monitors its communication channels to other processes for terminated processes at the other end and exits as a consequence of such terminations being found.

3. Previous work

One of the authors has previously compared the performance of weather model algorithms on a variety of parallel computing platforms[7][8]. Two different methods were used to solve the shallow water equations and these methods have quite different message passing requirements when run on a distributed systems. The first method uses finite differencing which requires only localised message passing whereas the second uses spectral methods which requires global message passing. Further analysis of the spectral code though shows that for an array of processes, data is only required to be passed across a row or down a column.

In a previous paper[9] the authors discuss the porting of the finite difference version of the weather program to a transputer system from a shared memory machine simulating a distributed system using the Argonne macros for message passing constructs. This work has also been done for the spectral method and on the Fujitsu AP1000 cellular array processor. The method used was to implement those Argonne macros required on the transputer system and on the AP1000. It was found that this was not a trivial task as the configuration issues made the task complex as the Argonne macros are for generalised message passing.

In another paper by two of the authors [10], the Helios Farm Library is discussed. This provides for simple implementations of distributed algorithms and can be used for master-slave models provided there are no data dependencies between slaves. There is no provision in the Farm methodology for worker to worker communications as all communications must pass through the centralised controller. Whilst it would be possible to implement finite differencing and spectral methods for grids using the Farm Library on an array of transputers, its performance would obviously be bandwidth limited

Coding of parallel programs tends to be done from the ground up every time a new program is written. Because the parallel constructs are at a low level form an analytical point of view, there does not tend to be carry over from one program to another. Parallel programs tend not to be modifiable, extendable nor robust in terms of avoiding deadlock when modified. Cut and paste methods don't tend to be productive with parallel code, especially with distributed systems.

The authors have presented a further paper[11] that abstracts the task of creating a finite difference application on an array of processors. This was the original version of BABBAGE and provided the correct sequencing of master-slaves processing as well as high level support for sharing of edge data between adjoining processes. It was soon realised that the sequencing was not generic to just finite differencing but to master-slaves applications. With suitable high level support for slave sharing of data the package could be used for other numerical methodologies. This versions adds the ability for slaves in the same row of the grid to

efficiently exchange their columns of a data and for processes in the same column to exchange their rows. These data sharing procedures are used in spectral methods where fourier transforms require data in a complete row or column.

4. The Master-Slaves Model

The master-slaves model is an efficient method for implementing grid based problems where the same operation is to be applied at each grid point. The grid is shared between slave processes whereas the master is responsible for overall control of the task. The task is distributed between the 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 providing synchronisation between the slaves. The slaves receive their portion of the grid from the master and perform the grid operation upon 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.

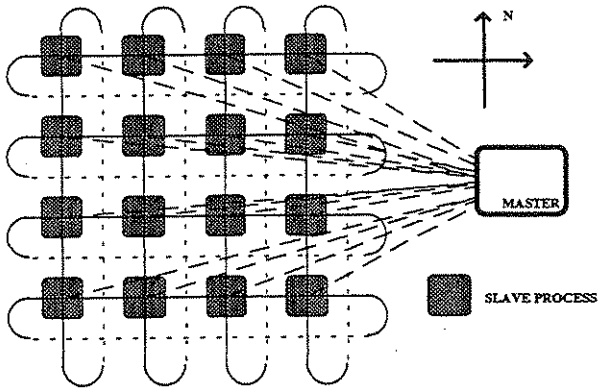

Fig 1. Master-Slaves configuration using a 4 x 4 torus

In figure 1, the solid lines represent the interslave communication channels whereas the long dashed lines represent the master-slave channels.

Because the slaves are equivalent, there is a lot of determinism in the master-slaves model. (Indeed it really is a SIMD computing model rather than MIMD). A generalised distributed algorithm for master-slaves application is in Figure 2. The steps in this may be classified as those that are generic to the master-slaves model and those that are specific to the particular application. Table 1 shows that generic steps whilst Table 2 shows the application specific steps. In creating an application using BABBAGE one need only create the steps in Table 2 as the generic steps have been already coded. To facilitate the data movements required in the application specific steps a variety of high level macros have been included in the software tool.

The master-slaves algorithm as in Figure 2 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 and the work routines represent the work to be done for one iteration only.

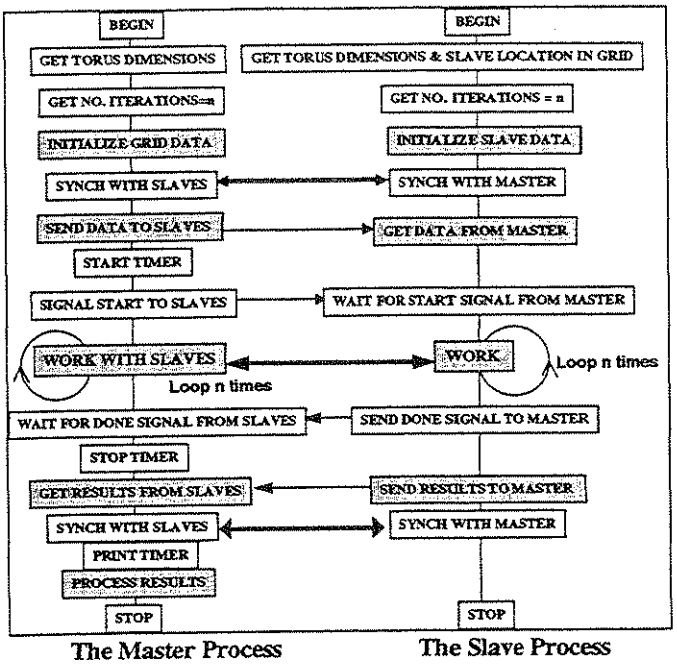

Fig 2. The generalised distributed algorithm for finite difference equations.

An example script is given in Table 3 for a heat flow application on a rectangular plate. Note that the one script is used. This undergoes separate compilation for the master and slave processes and is bound with the precompiled generic code. These processes are bound into a Helios task using the CDL compiler. Note that the Scatter and Gather functions are used in the common area of the script as they have the same syntax with BABBAGE in the master and slave but when compiled are implemented differently in the master and slave.

MASTER	SLAVE
Get Torus Dim.	Get Torus Dim. & Slave Locn.
Get No Iterations	Get No. Iterations
Synch with Slaves	Synch with Master
Start Timer	
Signal Start to Slaves	Wait for Start Signal
Wait for Done Signal	Send Done Signal
Stop Timer	
Synch with Slaves	Synch with Master
Print Timer	
Stop	Stop

Table 1. The generic steps for a master-slaves application

MASTER	SLAVE
Initialise Grid Data	Initialise Slave Data
Send Data to Slaves	Get Data from Master
Work with Slaves	Work
Get Results from Slaves	Send Results to Master
Process Results	

Table 2. The application specific steps

```
/***** Global *****/
#define h 0.1
#define hsq 0.01
#define dt 0.1
#define XWRAP FALSE
#define YWRAP FALSE
/* NX and NY are the dimensions of the data grid */
#define NX 64
#define NY NX
/* XX and YY are the dimensions of the processor grid */
#define XX 4
#define YY 4
/* temp is the data type of u */
#define temp 97
float u[NX][NY];/* The grid data structure */

/***** General *****/
sendData(){
  Scatter(u, u, temp)
}

getData(){
  Gather(u, u ,temp)
}

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

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

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,temp)

/* Perform finite diff for all 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() */
```

Table 3. An example of a finite differencing script using BABBAGE

5. The Macros in the BABBAGE software tool.

- Currently the task level data communications calls are implemented using macros. A future version will bind them into a library. It is also intended to encapsulate them as methods in an object oriented language version.
- The macros are based upon a further set of macros that have been written at level two of abstraction. These simplify synchronisation and configuration as well as providing some abstraction of data movement.
- The macros assume that the full grid data structure is declared in both the master and in each of the slaves. A future version will provide for an option that declares a reduced grid within slaves allowing for much larger grids to be manipulated.

5.1. Macros for moving data between the master and slave processes

These macros send the grid to slaves and collect it back without the need for the user to be concerned with connectivity.

Scatter(masterds, slaveds, messagetype)

- sends masterds in master process to slaveds in slave processes
- each slave receives only its share of the master data structure
- messagetype is a simple integer used to check the integrity of the message passing.
- user may wish to create a set of message type macros

Gather(masterds, slaveds, messagetype)

- collects each slave's domain as slaveds and returns it to the master in masterds
- messagetype as for Scatter

5.2. Macros for moving data between slave processes.

These macros are at the task level. which means that when used in the slave process function, all slaves perform the same data communication simultaneously. They both send and receive data and so effect complete communication of data from equivalent regions within slaves to equivalent regions within target slaves. Note that deadlock is avoided by having adjacent processors scheduled so that one sends whilst the other receives and then they swap roles. This should be faster than buffering message passing to avoid deadlock.

5.2.1. Edge macros

These are used in finite differencing where calculations at each grid point only depend upon nearby values. Before computations begin for each iteration, slaves must get data from nearest neighbour slaves for grid points at edges within slaves' domains. The edge macros copy the edges of slaves' domains as entities to a region just outside the opposite edge of adjoining slaves. If Edgewrap is turned on then the grid is treated as a torus and slaves at the extremities of the grid communicate with slaves at the opposite extremity of the grid. If Edgewrap is turned off then slaves at the extremities of the grid do not send edges that are at the edge of the grid.

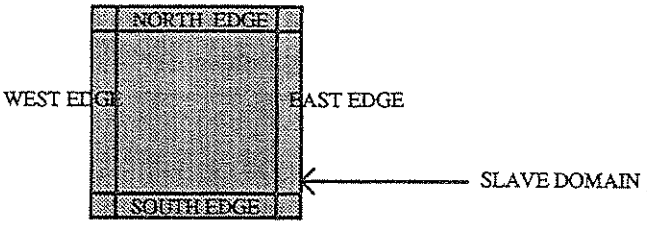

Figure 6. Sources of outgoing edge data.

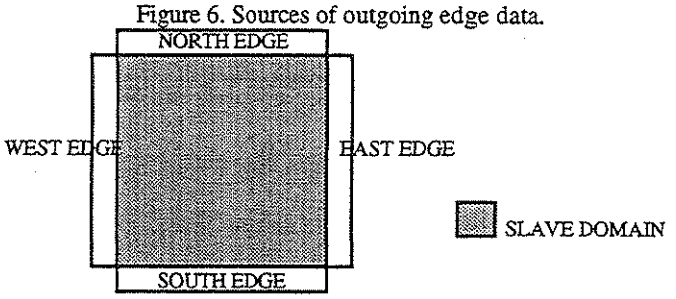

Figure 7. Destinations for incoming edge data

SendAnEdge(Direction, ds)

- where Direction ∈ [NORTH, SOUTH, EAST, WEST]
- Sends the data at the edge of each slave's domain.
- The edge is that in the direction specified and is sent the nearest slave in that direction.
- The width of data sent is determined by the globally defined symbol EDGEWIDTH. (Normally one).

Also

- Receives an edge from the opposite direction from nearest neighbour in that direction.
- Places it in edge area outside slave's domain in that direction.

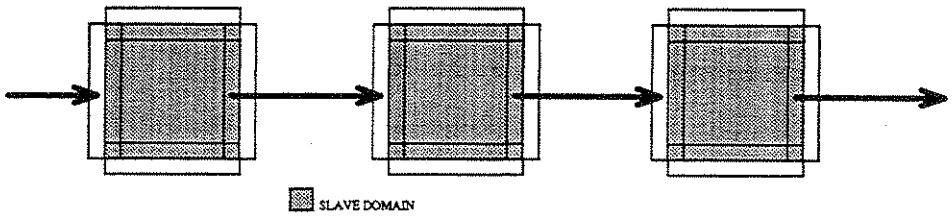

Figure 8. SendAnEdge data movements

SendEdges(ds)

- Sends all 4 edges to nearest neighbours and gets their edges in return.

5.2.2. Row and column macros

These are used in spectral methods where complete data in a row or column is needed for fourier transformation of a grid subregion. The row or column functions as a ring for passing data packets. These macros start with each slave passing a data packet to the nearest slave to the left in the row or column. This slave makes a copy and passes it on in the same direction. This continues until all slaves in the row or column have a copy of all packets sent in the initial step.

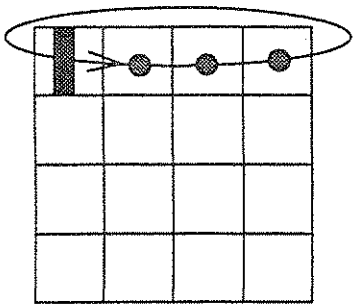

Figure 9. Passing data around a row

SendARow2Col(row, ds)

- Each slave sends the specified row within the slave's domain to all slaves in the same column.
- The row number is relative to the slave's domain. Hence $0 \leq$ row $\leq$ No. rows in each slave.

SendAllRows2Col(ds)

- As per SendARow2Col but all rows in slaves' domains are sent.

SendACol2Row(col, ds)

- Each slaves sends the specified column within the slave's domain to all slaves in the same row.
- The column number is relative to the slave's domain. Hence $0 \leq$ col $\leq$ No. columns in each slave.

SendAllCols2Row(ds)

- As per SendACol2Row but all columns in slaves' domains are sent.

5.3. Macros for collecting summative data

When issued in the master process and matched by a corresponding call in the slave process with exactly the same syntax, these macros return the current summative value for the whole grid despite the fact that the grid actually is residing in the slaves during computation. The slaves return their summative value to the master which merges them all to determine a collective value. The Helios implementation makes use of the vector library in version 1.3 of the operating system to efficiently extract the information from the slaves.

GetAv(ds), GetMax(ds), GetMin(ds)

- returns the current average, maximum or minimum value respectively for the whole grid data structure ds.

GetAmax(ds), GetAmin(ds)

- returns the absolute maximum or minimum values respectively.

GetSum(ds), GetSumsq(ds)

- returns the current total of ds or sum of all of the squares of values in the grid ds respectively.

Conclusions

BABBAGE is a software tool for implementing master-slaves configured grid based applications onto distributed computing systems with which two dimensional arrays of processors can be simply mapped onto the processors' topology. The software tool requires coding only at a task level to create an application. This makes implementations simple to produce, portable and robust. Macros are provided to effect data sharing for finite difference and spectral methods as well as for collection of summative values of the grid.

References.

[1] The helios Parallel Operating System, Perihelion Software Ltd, Prentice Hall

[2] ANSI-C Toolset, INMOS.

[3] All-to-All Personalised Communication on a Wrap-Around Mesh,Horie & Hayashi,PB-1, Proc 2nd Fujitsu-ANU CAP Workshop ANU, Nov 91.

[4] CAP-II Libaray Manuals [Hosat &Cell], Fujitsu Laboaratories, 2 Ed, Jan 90.

[5] Portable Parallel Programs for Parallel Processors, Boyle et. al., Holt Rinehart and Winston.

[6] PVM 3.0 User's Guide and Reference Manual, Geist et. al., Oak Ridge National Laboratory.

[7] A Study of Shallow Water Equations on Various Parallel Architectures, Abramson Dix and Whiting, Aust. Comp. Sci. Comms, V13 No.1, Feb 91.

[8] Super Computing Applications in the CSIRO-DIT High Performance Computation Project, Abramson et. al., Proc 3rd Aust. Super conputer Conf, Dec 90.

[9] Porting Parallel Programs from Shared Memory Multiprocessor Systems to Transputer based message passing systems, Jones Abramson & Hulskamp, Transputer Aust. 2, IOS Press, Amsterdam.

[10] Helios multiprocessor farm libary, Jones & Hulskamp, TAPA-92, IOS Press, Amsterdam.

[11] Tools to facilitate the implementation of grid based finite difference algorithms on distributed systems, Jones Hulskamp and Abramson, Proc WTC93, Sept 93, IOS Press, Amsterdam.

PCAT–93

PARALLEL COMPUTING AND TRANSPUTERS

edited by David Arnold, Ruth Christie, John Day and Paul Roe

IOS Press

Amsterdam • Oxford • Washington • Tokyo