Super Atari

by Mike Charnley Fisher

COMPUTER SHOPPER FEB 89

The Atari ABAQ has been been superseded by the ATW, the Atari Transputer Workstation. Mike Charnley Fisher looks at parallel processing under Helios on this new supercomputer

THE ATW (or Abaq as it was known) has already been described in some detail in previous articles (Issue 8, October 1988). Very briefly the ATW is a high performance, low cost graphics workstation. Its claim to fame lies in the fact that it uses the INMOS T800 transputer as a central processor. As you should all know by now, the T800 is a 32-bit processor rated at 10Mips/1.5MFlops, ie very fast compared with Intel/Motorola offerings. The unique feature of the transputer family of processors is that they have inbuilt serial communication channels, each of which is able to operate at about 2.5Mbytes/sec concurrently with the CPU (and onboard floating point unit). The T800 has four such 'links' and this makes it ideally suited to parallel processing applications. Internally, the ATW is able to hold 13 of these (along with anything up to 64Mbytes of memory – using 4Mbit chips) and, through two buffered 'links' can address any number externally. This gives the ATW the capability of being a 130Mip supercomputer in its own right as well as the ability to act as a gateway to even more powerful boxes such as Perihelion's Polyhedra (480 Mips+). To put all this in perspective the Cray YMP supercomputer is rated at 1000 Mips (on a L/mip basis it is an order of magnitude more expensive).

However, the new technology lying in the ATW is not limited to the transputer. A graphics blitter, now named Blossom, helps the ATW drive 4 graphics display modes at very respectable speeds (See the figure 'Blossom pixel power').

On entry level machines, to allow for these resolutions, there is a dedicated Mbyte of dual ported Video RAM and 4 Mbytes of user RAM. Along with hardware to support a SCSI interface (giving access to a wide range of hard/optical disks and tape drives), an inbuilt 40MByte hard disk, and of course the Motorola 68000 based I/O controller (in the form of a modified Mega ST board) the ATW represents remarkable value for money at the £3-4,000 price.

The machine on which this article is based is an early 'Issue 3' machine fitted with a Rodime 3.5" 130MByte SCSI hard disk and a transputer 'farm' card with just one extra T800 with its own Mbyte of RAM. And it is well worn – something like 1,800 hours of continuous use to date. Machines are shipped with the Helios operating system (about which this article is mainly concerned) and X-Windows, the rapidly emerging standard graphics front end for workstations.

Parallel Processing – In Context

The term 'parallel processing', apart from rapidly becoming a fashionable buzz-word, is applied to a wide range of different areas of computing. On the one hand it is applied to the 'very fine grained' instruction level parallelism typified by the working of the brain and other nervous tissue. Here thousands of very simple 'instructions' are processed at the same time. In machine terms the ICL DAP computer, and recent 'neural newtork' machines are closest to this type of computing.

A step up from this is the 'fine grained' algorithm level parallelism typified by recent approaches to analysing large matrices or arrays in finite element work. Also applied to graphics operations such as rendering (amongst others), this is the level at which the design of the transputer was initially targetted. On the transputer Occam remains the best approach to attacking such problems although in other environments there are other languages (such as dialects of Fortran) which operate at a similar level. This is an area where 10s and 100s of parallel operations can be seen to be of benefit.

However, it is the next level of parallelism, 'coarse grained', which has the widest level of applicability. It is also the level at which Helios and the 'parallel' languages are best suited. Here applications are broken down into 'tasks' and these tasks run on separate processors (making use of between one and, say, 20 processors). Conceptually it is easy to picture the use of a limited number of processors for any application (and of course even easier to imagine running multiple concurrent applications). Just consider, for example, a DeskTop Publishing (DTP) package. One processor can be running the 'operating system' task, another the 'window manager' task, another the 'font managment/display' task, another the 'spelling checker' task, and another the basic 'editor' task – already five processors used to create a superfast DTP environment.

It should be clear from the above example that 'parallel' processing is already all around us – designers of operating systems, particularly multi-tasking ones such as Unix, have had to worry about 'concurrent' processing for ages. The only difference is that now – instead of time-slicing on a single processor – multiple processors may be used. The one added complication is interprocessor communication.

Blossom pixel power

1280 x 960 pixels – 16 colours out of a pallette of 4096
1024 x 768 pixels – 256 colours out of a pallette of 16.7m
640 x 480 pixels – 256 colours out of a pallette of 16.7m
512 x 480 pixels – 16.7m colours out of a pallette of 16.7m

Communication – The heart of the matter

In a traditional environment such as the Dos or Unix world, concurrent tasks usually communicate through shared memory. Because each task has access to the same registers and the same address space it is relatively easy to display information on the screen or get information from the keyboard for example. Provided they are memory mapped, the devices can simply be accessed through pointing to the correct area of memory.

In the multi-processor environment things are a little bit different. In the case of the ATW the keyboard is logically connected through the 68000. In order for the main transputer to get a keypress it has to be transmitted along a piece of wire connecting the 68000 world to the transputer. If the application is running on another transputer the first transputer has to pass the key-press down yet another wire, and so on until the destination transputer receives it. The same is true when putting information on the screen. In the case of the ATW, screen memory is only accessible through one processor. If the application is running on a remote one then somehow or other signals have to be sent down wires to get the right processor to poke the right location in memory.

The problem is further complicated when software portability is considered. Logically speaking it should not make any difference whether an application is running on one processor or 10. The same piece of code should be transportable across all processor configurations if it is to be portable – how else can it be sold without giving away the source?

As it happens, the solution to these problems is partly addressed by the transputer architecture and micro-coded instructions and partly by the operating environment. Low-level communication is basically managed by the transputer itself. Managed as a byte stream, a transputer process will not send data to another process unless they are both ready – the programmer does not have to worry about low level synchronisation. In addition the transputer instructions for communication on-chip are identical to those for communicating through a link – the links are memory mapped. This means that at a low level it is possible to design systems which will multitask on a single processor and which will run across seperate processors if instructed – without change to the low level code.

Under Occam the high level communication problem is handled through logical 'channels'. The programmer explicitly declares which parts of the code may be run in parallel and explicitly opens channels (similar in most respects to 'pipes' in the

Continued on page 60

Continued from page 57

Unix or Dos world). Unfortunately there are a couple of weaknesses with this approach which have limited the success of Occam. The first is that it is up to the programmer to generate a harness to match physical resources to logical requirements – this has to change depending on the hardware configuration. The second is that once a physical link is allocated to an application it cannot be used (without extensive programming) by another application. Both of these aspects limit the ease with which it is possible to use Occam in the general purpose programming environment.

The way in which both the 3L 'parallel' languages and the Helios operating system get round this is through the use of a higher level of communication – message passing. The advantage with messages is that they are able to be sent between different locations simply by changing address and that they can be sent in packets – ie a number of different messages may be sent down the same physical channel.

In addition it is possible to get messages between two different locations by different routes – configuration dependency is largely removed. Messages also provide a convenient mechanism for introducing a degree of fault tolerance within a system. If a message doesn't get through, the waiting process simply sends a request for it to be sent again – it doesn't lock up.

The actual theory behind message passing is well documented and has been applied to operating system design for many a year (Tanenbaum's Minix uses message passing). The trick behind both the 3L languages and Helios is the way in which they both ensure that messages get routed correctly and without getting mixed up, without the programmer intervening.

The only real difference between Helios and the 3L approach is that Helios does it in an operating system, 3L within the language itself. Since the 3L method requires non-standard inclusions in the code, my own preference is for the Helios approach. Apart from the additional facilities provided by a full operating system it also lends itself to faster adoption by other languages. (I have not used the 3L languages).

Helios – The operating system

Helios is a fully-fledged operating system which provides all the facilities found under Dos, etc. Unlike Dos, (but like Unix) Helios is multi-user and provides the security controls necessary for this mode of operation. Languages now available under Helios include Ansi C, Fortran-77, BCPL, Modula-II, Pascal, Occam under the Inmos TDS, Basic, and an AI language called Strand.

Over the next few months Lisp, Ada, and Prolog – among others – are expected to become available. At a system programming level a 95 per cent complete version of the POSIX defined Unix library is implemented along with a complete clone of the Unix csh shell and all the standard Unix utility programs and facilities. At a lower, Helios-specific level, support is provided for message passing, list and queue handling, and semaphore control. Access to graphics is provided via version 11.2 of X-Windows – itself a system based around message passing.

As an operating system Helios has the advantage that it has been designed using latest software technology without problems of backward compatibility (unlike OS2). The most notable external evidence of this is in the size of the Helios nucleus (it needs no more than 100K to run) and the modular nature of the nucleus programs.

Very briefly, the nucleus consists of a kernal (containing memory management, the message handling, semaphore control etc) which is about 15K of transputer assembler. As this is the only part of Helios tied specifically to the transputer there is no reason why Helios could not be ported to other hardware environments should the requirement arise. Also in the nucleus is the system library. This is responsible for creating and accessing the 'Objects' and 'Streams' which are used to represent all Helios components from files, through executables, to processors themselves – all objects may be accessed with the same system calls. The remaining tasks consist of the Server library – which provides access to external 'objects' to the processor; the Loader – responsible for 'loading objects into memory; the Process Manager – responsible for controlling access to competing processes; and the Link Guardians – responsible for controlling access to the processor across the links. These programs represent the core Helios and they must be present on all Helios nodes. Raw transputer nodes – for use by Occam – are available under Helios through a TDS gateway. This allows 'coarse grained' Helios applications to control 'fine grained' processes better suited to the Occam model.

The flexibility of Helios and the distributed nature of the operating system is evidenced by the way other parts of the system appear. There is a RAM disk manager (which only goes on those processors needing a RAM disk), the window manager only gets loaded on the processor which has a console attached, and the printer drivers only get loaded on those that have a printer attached, etc, etc. This is in direct contrast to how Unix and Dos operate – where everything is loaded – regardless of where it is used.

This concept also applies to the language support libraries. Separate modules exist for 'C', the floating point routines (different, depending on whether the floating point T800 or integer T414 is being used), and the POSIX library. These only get loaded where needed, a fact which is common to all language support libraries. Furthermore, once loaded these become available to all applications using the library – they are 'shared'. In practical terms this means that the executable size of applications is generally a lot smaller than the Unix or Dos equivalent. For example one application – written in Fortran – has an executable size of about 30K under Helios.

On a SUN running UNIX the same executable is about 130K.

None of this is any good without a degree of stability, and while it is true to say that Helios is not without its early bugs it is certainly a lot more stable than expected. (Reports to the contrary arising out of COMDEX in Las Vegas were due to problems with the first Issue 4 machines to appear in the field – they were very over-sensitive to static electricity – a problem now being resolved). As evidence of this I have personally put about 4 MBytes of 'C' through the Helios 'C' compiler and about 5MBytes of Fortran source through the Meiko/Top Express F77 compiler running under Helios (which itself uses BCPL). I have also been using X-Windows as a working environment for the last couple of months.

Aside from the degree of compatibility with existing Unix systems, the part of Helios which is of most interest is its ability to handle parallel forms of comp[illegible]tation.

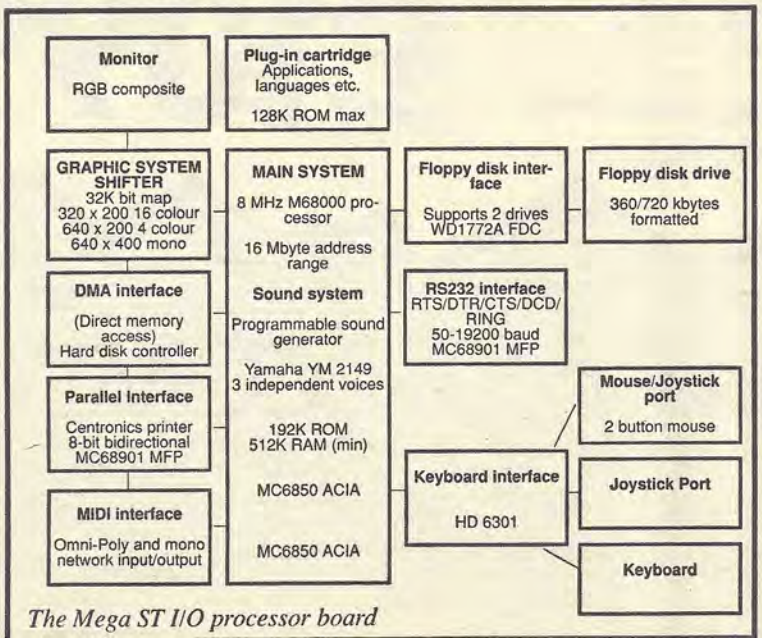
Multi-threaded routines under Helios

Like OS2, Helios provides support for multi-threaded code. In Helios terms multi-threaded code is different to multi-tasked code in that the former usually involves hiving off a function within the executable to run as a separate process on the same processor. It is thus able to share variables and address space without resorting to message passing.

The Helios system level calls to achieve this are appropriately named NewProcess(), RunProcess(), ZapProcess(), and a combination of the two former – Fork() (not to be confused with Unix/Posix fork()). These calls allow the programmer to nam[illegible] any function as being able to run in parallel with the rest of the application – it does not have to be a separately executable module.

The most common use of such a construct is to provide an event handling mechanism within an application. Consider the case where an application is waiting for a mouse button press OR a keyboard press. Using standard programming constructs both devices have to be polled (not possible using ANSI standard 'C') or the program has to wait for one or the other, but not both at the same time – and the program locks while waiting.

Through two calls to Fork() and by setting up suitable semaphores to ensure that device events do not get confused it is possible for the application to monitor both devices at the same time and, if nothing is happening on either, for it to go away and do something else until something does happen. Under other operat-



The Mega ST I/O processor board