Tide

12 May 2004

Abstract

A method for information interchange on tightly coupled massively parallel procesors is depicted. The basic architecture of a distributed micro-operating system is described, which accomodates for any parallel computing paradigm on high volume arrays of low cost processors.

1 Discussed environment

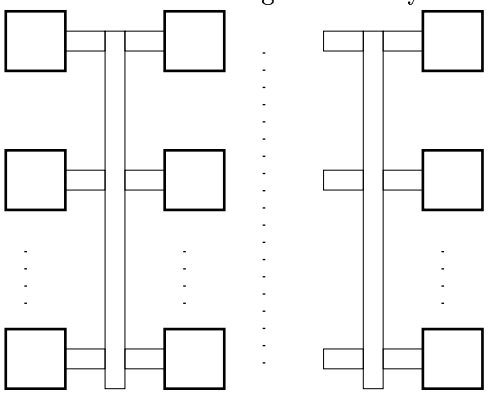
Information transmission is a critical part in multiprocessor arrangments as is the choice of tecnology at each procesor node. In this document we discuss higly symmetric multiprocessor systems distinguished by the following caracteristics:

- each processor provides just minimal processing and I/O functions - we use preferably an array of tiny CPU-cores on one silicon chip or PCB-board
- each processor counts with a small amount of RAM and ROM, eventually with very high speed access.
- each node can be constructed by one processor, or in turn by another array of several cores.
- the nodes are directly interconnected with high speed data links/busses. The used interconnection topology does not play the principal role in the discussion.
- The array has medium to high speed access to similar amounts of conventional RAM found in todays workstations
- Input/Output functions of the array are taken over by the edge nodes, either via direct comunication with onboard I/O controllers or with specialized procesors which interact with the array via it's interconection busses.

This environment is inspired by the availability of very small, high performance, low cost CPU-cores, which are easy to multiply on one chip.

Standard symmetric multiprocessor aproaches, require complex hardware to manage shared access of several CPU's to shared RAM, which converts main

Figure 1: Array of independent CPU's



memory in a bottleneck and raises higly complicate sincronization issues partially dealed with in the hardware.

On the other hand, in distributed systems, serveral independent high performance computers are connected together via a high speed LAN, each of them running their own operating system. In either case, the CPU's involved use wide address busses and again highly complex hardware acceleration mecanisms: floating point units, n-level memory caching, memory segmentation and paging, etc. resulting in high per unit costs in both hard- and software.

We hope that this new approach, which combines very larg amounts of "dumb" CPU's in combination with new programming approaches, can compete in price, security and performance with the standard solutions actually deployed.

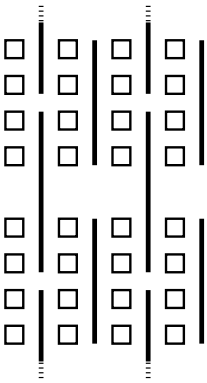
2 Topology

In massively parallelized systems designers try to yielding low distances between nodes in the networks, to maintain low information transmission times with increasing numbers of CPU's. Our approach does not enfatize network topology. Each compinations of CPU and transmission tecnology yields good comunications performance up to a certain number of interconected units. The hardware designer should choose a good guess to form clusters of nodes which interconnect tightly with high performance. The clusters in turn would interconnect between themselfes with high cluster-to-cluster performance links.

The image shown in the previous section suggest, for example, that each node is conected to two busses. so that West-East propagation of Information would have to be routed via n clusters while North-South propagation would be handled by the direct bus connection. The following picture shows how to expand this approach to huge dimensions, without saturating the busses, at the cost of increased number of hops between distand nodes.

The figure shows Clusters of eight CPU's, each of which has two connections,

Figure 2: Grid of CPU clusters



one to a West-Side bus and one to an East-Side bus. Let us observe the top rightmost four CPU's which are connected to one bus on their East-side. The upper two CPU's share their West-side bus with a North-Side located cluster, while the lower located two CPU's share their West-side bus with a South-Side located cluster.

As we argument later, locality of code and data will not be exploited by caching, but by automatic grouping the code around CPU's of nearby clusters.

3 Factorization and Computing Quants

In a first approach to parallelize programs we asume, that the algorithms used to realize some processing task can be - and are - chosen in a way, that very high factorization can be acchieved. The "main" program and all subroutines can be executed individually in any CPU on the array, because they fit into the (small) available memory of each. We also suppose, that each subroutine only has to process a very small amount of data as parameters and returns a very small amount of data as results. At runtime the subroutines and the parameters are packed together into a *quant* and sent as a messages to the array. They are transfered either as a block of data, or by reference to their location in the (yet to define) secondary storage. The quant is distributed as a decaying wave to the whole array, i.e. it is stored and replicated through the grid until it gets *absorbed* by one or more CPU. We will deal later with the implications of decay and absorbtion, as well as with details of the messaging process.

The originating CPU leaves it's fingerprint in the message, by which the result can be routed back through the grid. A CPU which decides to execute a quant, instead of replicating it is said to *absorb* it. During the execution of the respective code, further quants are eventually spawned to the grid from the CPU which absorbed the original quant during execution. As a result, the execution of a specific program results in many superposed waves of computing quants which dissapate back and forth trough the whole grid as computation

advances.

4 Secondary Storage

To hold the collection of subroutines, as well as global data and parameters which consist of large volumes of data a linuar virtual memory space is used. The secondary storage to hold the virtual memory is conventional RAM built into the computer. The algorithms used to allocate and assign memory without conflicts and race conditions are well known from the theory and practice of operating systems and are not discussed here. The virtual memory allows as to broaden the range of algorithms that can be executed by relaxing the restrictions of the former section: Instead of passing parameters literally and whole sections of replicated code, pointers to them can be packaged into the quants. If a CPU absorbs a quant, it has to fetch the block of code from virtual memory and located in the secondary storage. While this can result an sending forth and back several messages through the grid to the CPU's concerned with I/O, there is always a chance that the required code is already present in another node in the message path, and can be either fetched directly from there. It can also be decided to execute the quant in that node if it is available.

The virtual memory should be designed sufficiently large to accommodate *all* information ever needed. A fourty bit address space, for example could address more then ten terrabyte, which should suffice to hold most programs and data people need for day to day use. To simplify access to secondary storage it is perceived as paged memory, which is partially cached in conventional RAM-chips and periodically swapped/backed up to harddisks.

If a conventional filesystem is required it is advisable to store it on a simulated RAM-disk in the virtual memory, however we assume that this complication is not required. One function of the required meta-operating system would be to guarantee that the grids state in local and conventional memory is completly saved to disk, and restored on power up, so all data processing can be done in memory.

5 Last resort: emulation

To tackle with un-factorizable computing problems it is perceivable, that a virtual machine can be programmed with only the given components in away that it would be able to resolve such tasks. Of course the processing would not be as efficient and it would have to be shown, that the mayority of computing can be sufficiently factorized to proof our concept feasable.

It is to expect that there will be a great number of emulated machines, in the first place to take advantage of existing non-parallelized computer programs and systems. On appealing example would be the emulation of a Sparc or G4 Risc processor including the corresponding environment to be able to run Solaris, OS-X or Linux on top of the array. However a shift of the actual programming

and computing paradigms could obsolete these systems anyway.

6 Input/Output

A very simple aproach to handle the Input/Output system would be to use the "leftover" buses at the edges to connect directly I/O processors to them. On input events these I/O processors would directly create quants with the input data as parameters and inject them into the grid. For output it would suffice that they could receive messages with the respective data to send to the output device. It would also be possible to configure the edge CPU's of the array in a way that makes them aware that they are dealing with dedicated hardware. In this case they would privilegedly absorb quants concernd with programmed output. In any case, the grid should have a notion of "inside" and "periferical", so that at least routing paths of outbound messages are shortened and no unnecesary omnidireccional quant-waves be produced on the grid for them.

To illustrate the point, we bring an example for an input devce. Suppose we re-programm a conventional Keyboard driver chip (which is a complete CPU by itself) in a way that, when pressing a key, it generates a quant, which contains as a parameter the keycode, and as code a pointer to a routine in virtual memory, which in turn deposits the parameter savely in a keyboard input buffer. Instead of transmitting one or two bytes, it would send a block of data to the node/cluster it is conected to. The Keyboard driver does not need to have any notions of the code it transmits, they are hardcoded paquets of data interpreted by the machine code of the array CPU's.

It seems plausible, that input/output routines will migrate towards the edges of the grid, while mere computing tasks will rather migrate to the center. This seems desireable, because load spreading amongst several CPU's should be more efficient at the center nodes of a grid, then at the lesser populated edge nodes.

Bootstrapping and periodic maintainance tasks will also occure from the edge, the first in form of a Boot ROM injector which could inject code and data to place the basic routines of the meta-operating system into the virtual memory, and in form of a Timer/Interrupt processor which could inject periodically quants with maintainance tasks, like flushing the dirty blocks of the virtual memory caches to disk.

7 Two staged operating system

Two levels of operating systems are perceived:

- The micro-os, which is located on ROM inside each CPU-core
- The meta-os, which is located in the virtual memory/secondary storage

The former has to be as simple as posible and absolutely identical on each CPU of the grid to make it possible to be replicated many times on one chip. The

micro-os may have no "awareness" of the specific array configuration, it has the only function to provide interaction with the grid (neighbours) and a minimal amount of task administration.

Although we maintain the notion of inside/outside in the grid, the only information about the topology a node needs to know is how to reach it's neighbours. Whence the bootstrapping of the grid is provoced by an edge node (e.g. the mentioned Boot ROM injector), requireing maybe to fetch a block of code for a quant from virtual memory, it would suffice for a node to route the request-messages through the neighbour from which the (boot) quant has been delivered.

The principal tasks for the micro-os could be the following:

- Inbound comunication:
  - wait for a quant. If a quant is received:
  - calculate the decay, and decide if the quant has to be forwarded to some neighbour node(s)
  - decide if the quant will be absorbed. Some parameters which decide about absorbtion are: priority/decay status of the quant, cpu-load, availability of the code/data in RAM, availability of space to place aditional code into local RAM
  - if the quant is absorbed, schedule the quant for execution
- Execution
  - Execute the code of the currently active quant
  - If it contains other quants, schedule them for exposición to the grid
  - If it requires or modifies virtual memory obtain the respective buffers by either creating a quant or queuing a message
  - if the quant terminates, release resources, if parameters have to be returned pack them into a message and queue it
  - reschedule the present quants
- Outbound comunication
  - send queued messages and created quants to the grid (neighbours)

The task queue can be as small as one task, so that cpu-load would always be 0 or 100%. In dependence of the capabilities of the core it seems however feasable and desireable that the micro-os contains a tasking system, which queues/caches quants so that they can be reused/executed without delay in case a task blocks (creates a quant) or a quant's code is required by a neighbour or near node.

Messages can be completely emulated by quants or implemented independently. Quants can also work only with references to virtual memory or completely without references, consisting only of executable code and data, which

in turn can be pointers and code to retrieve them. However we suppose that the implementation of both options in either case provides valuable increase in flexibility and eficiency, of course with the cost of increasing the complexity of the micro-os.

The meta-os would have to deal with organization of the objects stored in virtual memory (administration, protection, accounting), bootstrap and shut-down, collection of routines for interaction with the "outside" world, and basic user interaction: command-loop, os-shell, console, and the like. We suspect that a variety of diverging concepts will be perceived for the meta-os.

8 Quants again

As stated earlier, a quant contains some data and meta-information:

- Decay-information
- Fingerprint of the generating node
- Parameters
- Code

The decay information is a value (i.e. an unsigned integer) which indicates the distance to the originating CPU. When a node receives a Quant it remembers the neighbour from which it was received and decides if it absorbs the quant, - i.e. if it will execute the code. In this case, the results (if any) are computed from the parameters (if any) and sent back to the originator through the remembered neighbour where it came from. If the node decides not to absorb the quant, it decrements the decay value and forwards the modified quant away from the neighbour (and it's ancestors) and away from the peers of them (other nodes which have received the quant from the same neighbour).

If there is no further node to send the quant to it hast to be "reflected" back to the grid. The lower the decay value, the higher has to be the probability for a quant to be absorbed by a node. If it reaches a certain threshold (eventually 0) the quant has to be absorbed or a restart action has to be taken, e.g. a message to the originating node that the quant has been lost.

Of course there can be developed numerous conditions where quant-waves can provocate storms, or deplete before they execute, i.e. the quants get lost. It would be the subject of further studies to provide inside how to prevent or aliviate or work around this situations.

Without further thinking it seems convenient to count with special decay information (values) to be able to:

- execute forcibly a quant, independent of the nodes availabilty, or to
- broadcast a quant without decaying so that it reaches forcibly all nodes of the grid.

It could also be interesting to specify mecanisms by which quants can be absorbed by various nodes in order to cache it's code or to improve the probability to return a result sooner than with the default distribution/absorbtion mecanism.

The fingerprint is used to identify the originating node. However the following algorithm would eliminate the need for the enumeration of the nodes:

- A node that generates a quant, adds a unique fingerprint to it: where unique is to be thought of in the node's point of view. The fingerprint is either a random or a secuencial number.
- Each node which forwards a quant stores the identity of the neighbour it received the quant from as well as the fingerprint, to be able to route an respective answer back.
- If a quant has a fingerprint already used by another generator, the receiving node alters the fingerprint with a value of it's own, stores the new value with the original value and forwards the quant.
- When a return message (or quant) is received, the fingerprint and route are looked up. If the fingerprint has been mangled by the node, it is restored before returning it to the respective neighbour.

This algorithm was designed out of the moment and has to be revised thouroughly.

9 References

The ideas presented in this document are primary inspired by the MISC processores designed by Charles Moore and the Forth programming language, which lends itself to massive code factoring and anonymous execution of code at very high execution rates. The principles shown however should not depend on a Forth/Stack processor but should be able to implement on any type of (micro-) processor arquitecture.

See the following links for references to parallel Forth computing:

- Interview with Ch. Moore about the c18 processor: http://slashdot.org/interviews/01/09/11/139249.shtn
- Chat log with Charles Moore (04/2002): http://www.ultratechnology.com/chatlog.htm
- Berd Paysans b16 processor array: http://www.jwdt.com/~paysan/b16-eng.pdf
- Homepage of colororth and VLSI chip design: http://www.colorforth.com
- Linda distributed memory implemented in Forth: http://www.ultratechnology.com/4thlinda.html