Parallel Computing on MultiSpert

Philipp Färber
*

TR-*TR-97-046*

December 1997

Abstract

This report provides an overview of the MultiSpert parallel computer system and its performance characteristics. In order to preserve the existing programming and runtime environments, we have straight forwardly extended the single board system in a master/slave architecture with the SUN host as master controlling up to 16 Spert-II slaves. We describe the underlying hardware and its limitations, as well as the additional communication layers, which provide an efficient remote procedure calling mechanism. Timing measurements on a 5-node prototype confirm MultiSpert scalability to high performance levels.

*International Computer Science Institute, 1947 Center Street, Berkeley, CA 94704

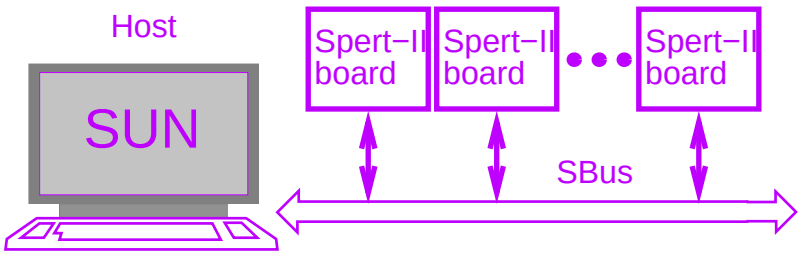

Figure 1: MultiSpert system

1 Motivation and Overview

The Spert-II vector-microprocessor [WAK+96] board was designed as a workstation accelerator to speed up artificial neural network (ANN) training in the context of a hybrid speech recognition system [?]. For this task, Spert-II surpasses the performance of today's workstation by factors of 3-10, allowing us to train large networks with high volumes of input patterns in the course of a few days, rather than weeks.

To further reduce training time, we have combined several Spert-II boards to form a small scale multi-computer, the *MultiSpert system*. In addition to offering high speed-ups relative to a single board, MultiSpert should take advantage of the existing programming environment with compilers, libraries and host interaction mechanisms.

Section 2 describes the resulting master/slave architecture, which straight-forwardly extends the performance of a single Spert-II by allowing for the concurrent use of up to 16 boards per host. Basic system limitations, raw bandwidth and peak performance estimates are investigated.

Section 3 describes the low-level software which makes the Spert-II boards easily accessible as regular SBus devices. The *SPC Spert Procedure Call* library implements an RPC-like mechanism to invoke remote Spert procedures and to facilitate the transfer of call arguments and results. Since the single master handles all data transfers across the shared bus, an application's computation to communication ratio is critical to its scalability.

Finally, section 4 discusses MultiSpert performance for different system sizes, applications and host architectures.

2 MultiSpert Hardware and Firmware

2.1 System configuration

Figure 1 shows a Multi-Spert system, consisting of a SUN Workstation and up to 16 Spert-II boards. The boards are double-slot SBus cards and may be plugged directly into the SUN host workstation. In order to increase the number of simultaneously attached boards, we use commercial SBus expander boxes, which connect 4 Spert-II boards to a single slot.[1]

The Spert-II boards contain the T0 vector microprocessor, 8 MB of SRAM memory, and a Xilinx FPGA managing data communication as described in the following section. The boards identify themselves to the host at boot time as regular SBus devices, which are managed through the *Spert device driver* (see section 3.1).

[1] Due to hardware errors in the expander boxes, we had to revert to only using two boards per slot.

2.2 Communication

The Spert-II board memory can be read and written from the SUN host via the T0 processor's *TSIP* byte-serial interface [AB97]. This interface reads 20 Bytes of address and data from the TSIP port at the rate of 1 Byte per cycle, before using one cycle of the T0 memory bus to transfer 16 bytes of data to the desired 32-bit address. With a 40 MHz clock, this results in a peak bandwidth of 32 MB/s. A Xilinx field programmable gate array (FPGA) translates regular 32-bit SBus memory write and read transactions to TSIP operations so that all of the on-board memory can be directly accessed from the host at random.

The data transfer rates achieved with the latest version of the Xilinx code, which implements neither read ahead nor burst mode transfers, are listed in table 1. A significant part of the difference between peak and measured bandwidths can be attributed to inefficiencies of the SBus protocol for single 32 bit transfers, especially when the additional level of SBus expander boxes is introduced.

Host	Sparc-2	Ultra-1 + expander box	Ultra-1 (direct)	T0
bus clock rate	20 MHz	25 MHz	25 MHz	40 MHz
peak bandwidth (R/W)	80 MB/s	100 MB/s (?)	100 MB/s	26/32 MB/s
measured Read BW	3.2 MB/s	3.22 MB/s	3.8 MB/s	
measured Write BW	5 MB/s	7 MB/s	10 MB/s	

Table 1: MultiSpert bandwidth limits

2.3 The TSIP Glitch

The T0 TSIP external interface was designed to work concurrently with regular vector memory operations, but we have discovered that data is occasionally corrupted when TSIP and vector store operations occur at the same time. Since the details behind this problem are not entirely clear, the following workarounds have been used:

- reduce the T0 clock rate to 33 MHz. Using this originally planned clock speed gets rid of the sporadic data corruptions and allows full communication and computation overlap.
- avoid any TSIP transaction when T0's vector-memory unit is busy. This means that the host has to synchronize with the Spert-II board before each data transfer to make sure that T0 is idle. This policy turns out to work well when several boards are used, since only the one board involved in the data transfer has to remain idle. Especially in the case of four boards, we have found the performance limiting factor to be the host transferring all data across the shared bus, rather than the slave's computation rate.

3 MultiSpert Low-Level Software

Several layers of low-level software support the MultiSpert system on the host side to make it easy to use. Figure 2 illustrates the different software layers, which are described in more detail below. The program statements on the right hand side show examples for the call interfaces between layers.

3.1 Spert Device Driver

Through the *Spert device driver*, the Spert boards behave like regular UNIX devices, which may be accessed with the standard system calls open(), read(), write() and ioctl(). At boot time, the driver automatically configures the attached boards and creates device links in the /dev directory to keep track of the different instances (see [**?**] for more details on SUN device drivers).

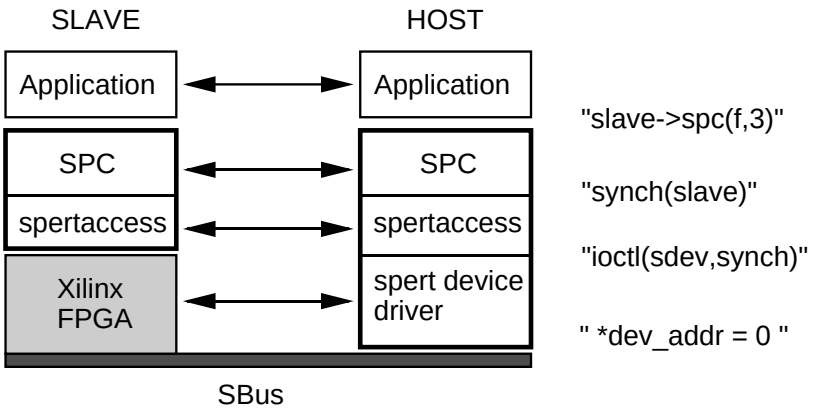

Figure 2: MultiSpert communication layers and interface examples

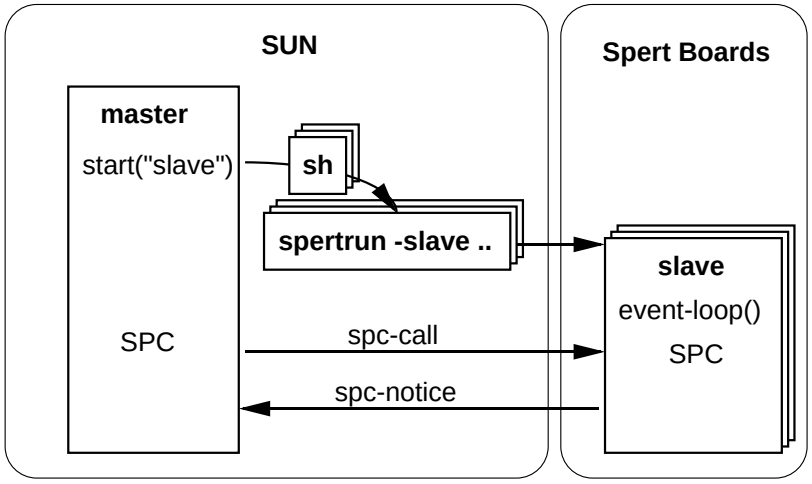

Figure 3: MultiSpert program structure at runtime

The device driver translates read() and write() requests to a sequence of SBus transactions, which adhere to the communication protocol implemented in the Xilinx FPGA on the Spert board (in the latest FPGA version, this simply consists of consecutive 32-bit memory read and write transactions). Through ioctl() requests, the device driver provides additional functionality, such as processor temperature control or access to internal registers.

3.2 Spertaccess

Although the device driver routines provide immediate memory access to the Spert slave, they don't solve the problems of starting remote program execution and of managing Spert memory. The *spertaccess* library provides routines to run Spert executables and allows the creation and transfer of communication buffers.

In order to take advantage of the existing runtime system on the host, the spertaccess routine "*start*" forks one instance of the "spertrun" program for every slave. As in the case of using only a single Spert-II board, each "spertrun" then handles the downloading and starting of a Spert executable, as well as the interpretation of system calls from the slave.

Figure 3 illustrates this program structure. The master calls the "start" routine for every slave,

which forks the spertrun program with the corresponding arguments (through a shell), which in turn initiates the desired Spert executable on the slave. From there, the master and slaves communicate through the SPC mechanisms as explained in the following section.

3.3 SPC: Spert Procedure Calls

Based on spertaccess and the device driver, the SPC library implements a remote procedure calling mechanism, which allows the execution of specific routines on the slave.

After creation of a slave instance, the host repeatedly sends call requests, consisting of an argument buffer and the call identification number. After an explicit synchronization, the result data may be read back to the host. The slave simply waits in an event loop until a call request arrives with the necessary arguments, then executes the routine and notifies the host after completion.

The commented program listing in figure 4 shows a simple example of a concurrent MultiSpert application. The master creates two slave instances, each of which immediately enters an event loop waiting for calls to the routines "sum" or "sumsq". The call arguments are written to the *spc_args* structure using the *put* method of the appropriate data type. The actual call then transfers this data plus the index of the desired routine to one of the slaves, which immediately starts computation. Since both routines use the same arguments, only one *spc_args* structure is required. As soon as the summation is finished on a slave, it notifies the master. In the example, the master issues two calls which execute in parallel, before waiting for the notification of slave number one. It then reads back the integer result, while the second slave may continue with its computation.

file 'host.cc'

```
main()
{             // create SPC servers
   spc_server *board1=new spc_server("xspert");
   spc_server *board2=new spc_server("xspert");

   int *a = read_array( 256 );
              // create SPC arguments
   spc_args *args = new spc_args();
   args->put_i32( SPC_IN, 256 );
   args->put_vi32( SPC_IN, a, 256 );
   args->put_i32( SPC_OUT, result );
              // start remote execution
   board1->call(IDX_sumsqr, args);
   board2->call(IDX_sum, args);
              // synchronize & read results
   board1->wait();
   board1->get_i32(&sum);

   board2->wait();
   board2->get_i32(&sumsqr);
   ...
}
```

file 'slave.cc'

```
sum(SPC_ARGS args)
{             // extract arguments
   int cnt = *(spc_getarg_i32(args));
   int *data = spc_getarg_vi32(args);
   int *res = spc_getarg_i32(args);
              // do computation
   int sum=0;
   for(int i=0;i<cnt;i++)
     sum += data[i];
              // store result
   *res = sum;
}

main()
{             // init call array
   SPC_CALL *calls = { sum, sumsqr };
              // start event loop
   SPC_EVENT_LOOP( calls, ARGBUF_SIZE );
}
```

Figure 4: Master and slave code examples

Sending the arguments along with the call request ensures that there is no communication/computation overlap at the beginning of Spert execution, so that the problem described in section 2.3 doesn't occur. For the same reason, the user has to make sure that the *get_result* calls, which read data back from the Spert board, never get executed before synchronization has taken place.

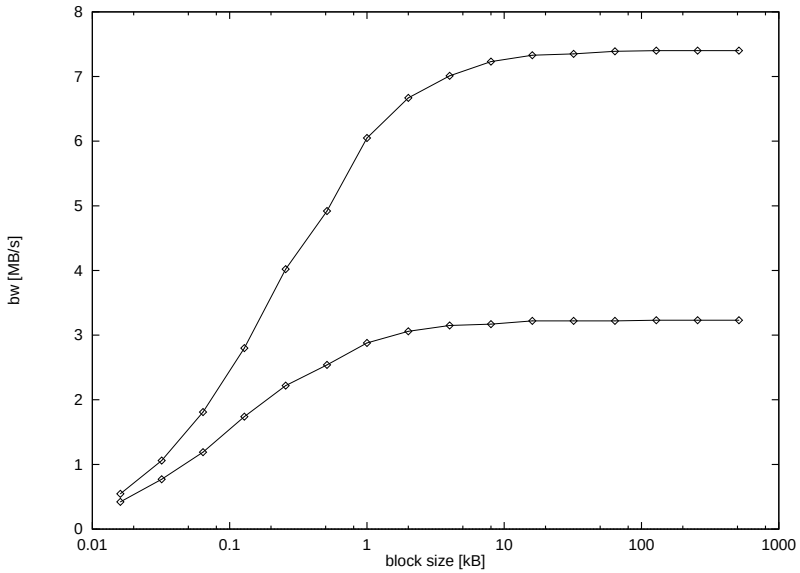

Figure 5: Host-Spert read and write transfer rates

4 MultiSpert Performance

4.1 Communication Performance

Figure 5 shows the data transfer rates obtained with directly calling the read() and write() routines of the Spert device driver. The lower transfer rate for small block sizes is due to system call overhead on the host. The numbers were measured for an Ultra-1 host with an expander box.

4.2 SPC call overheads

The SPC library introduces the call and synchronization overhead shown in table 2. These numbers were obtained using IPM on an Ultra-1 workstation as host with the Spert slave attached via an expander box. Although the mechanism is much more efficient than most remote procedure call implementations, very short remote routines may not be worth this call overhead.

SPC call (no arg.s)	25 μs
SPC call (with arg.s)	50 μs
synchronization	10 μs
read result	5 μs

Table 2: Overheads of SPC call, synchronize and result read

4.3 Single Board Performance

Since there is no overlap of computation and communication, the maximum performance which can be achieved with a each Spert slave is limited by the application's communication to computation ratio r_{CC}:

$$\text{achieved performance} \leq \frac{1}{1 + r_{cc}} \cdot \text{peak performance}$$

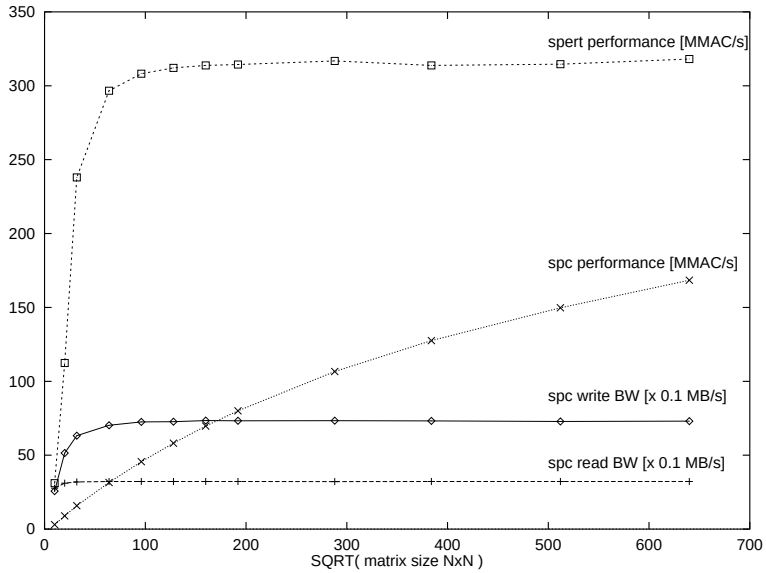

Figure 6: Single Spert matrix multiplication performance

For example, an r_{CC} of 1.0, i.e. 50 % of the overall time is spent on communication, means that each Byte of input/output data (transferred at 5 MB/s) keeps T0 busy for about 8 cycles (at 40 MHz).

The middle curve in figure 6 shows execution performance for an NxN matrix multiplication on one Spert using SPC. Although the T0 processor operates at its peak speed of 320 MMAC/s for relatively small matrices (top curve), communication cost (lower curves) significantly reduces the achieved performance. Since the number of multiply-accumulate operations increases with $O(N^3)$ while the communication time only increases with $O(N^2)$, r_{CC} drops linearly with N. An r_{CC} of 1.0 is reached for matrix sizes of about 500x500.

In order to keep the slave optimally busy, master and slave computation should overlap as much as possible. Data dependencies from one call to the next should be reduced through pipelining (see next section).

4.4 Multi-Spert Performance

When several Spert slaves are used concurrently, the single master host and the shared bus soon become the performance limiting bottleneck. Figure 7 shows timing diagrams for a typical slave routine which receives a vector of input data, performs a fix amount of computation per element, and returns a result vector. As more nodes are added (see top figure), the computation time per node decreases while the total amount of communication (to be sent across the shared bus by the master) remains about the same. Eventually, the bus saturates and master performs communication 100% of the time, while the slaves have to wait idle for data to be read and written (figure on right).

As illustrated in figure 7, the partial execution time for each of the P Spert slaves consists of the input and result communication times T_{in}/P and T_{out}/P, plus the node computation time T_c/P. When a routine is executed repeatedly, resulting in the call pattern shown in the central box, execution thus requires the total time $T_t = T_{in}/P + T_c/P + T_{out}$, unless the bus saturates, in which case $T_t = T_{in} + T_{out}$.

In order to avoid slave idle time, pipelining should be used (lower box in figure 7). Instead of waiting for all slaves to finish computation, the master calls the next routine immediately after synchronization. For the unsaturated pipelined case, total execution time evaluates to $T_t = T_{in}/P +$

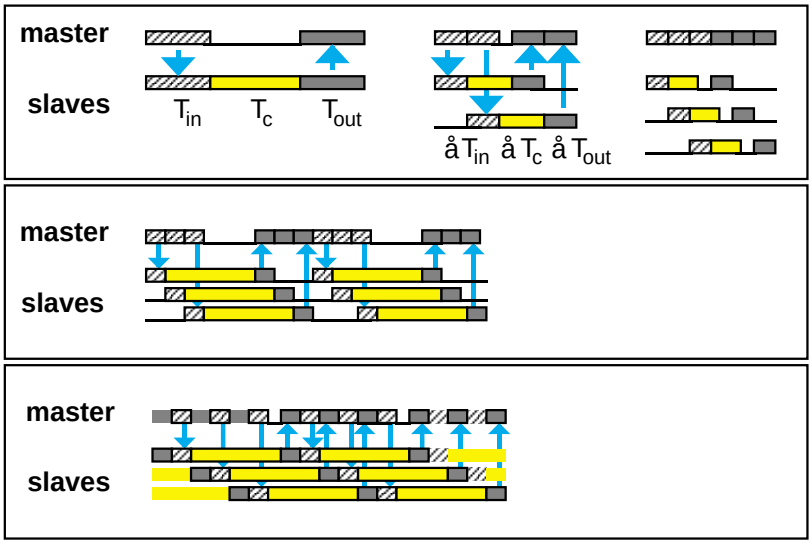

Figure 7: Concurrent SPC execution on MultiSpert

$T_c/P + T_{out}/P$, i.e. performance scales linearly.

For a different class of application which transfers the same input and output arguments to/from all slaves (i.e., simulating broadcast), communication time (T_{in}, T_{out}) increases linearly with the number of slaves. In this case, total execution time amounts to $T_t = T_{in} + T_c/P + T_{out}\dot{P}$ (or $T_t = T_{in} + T_c/P + T_{out}$ in the pipelined case), so that saturation occurs already for few slaves.

These simple performance models are illustrated in figure **??**, which shows speed-up graphs for different values of T_{in}, T_{out} and T_c.

5 Summary and Outlook

This technical report describes the MultiSpert system and its performance. The intuitive master-slave architecture allows up to 16 Spert board slaves to be attached to a single host.

Communication between master and slaves is supported through several layers of low-level software. Multi-Spert is easy to program using the SPC remote procedure call mechanism.

The System was then benchmarked and characterized in terms of communication and computation performance.

In order to further increase Multi-Spert performance, the following improvements may be considered:

- using burst mode Data transfers instead of single word accesses increases effective bus bandwidth by about a factor of two. Unfortuneately, the considered workstation does not offer explicit access to burst mode, so that this option was not further investigated.
- the use of an independent DMA bus master would free the host processor during communication.
- supporting a broadcast operation would greatly reduce the necessary communication time for applications, where input data is shared among all slaves. Since the regular SBus specification does not support a shared acknowledge line, non-standard hardware modifications would be required.

- In order to use the full SBus bandwidth, multiple bus masters ahould be allowed by supporting direct data transfers between Spert boards. In this case, the resulting model of computation would be a symmetric multicomputer, rather than a master-slave model.

References

[AB97] Krste Asanović and James Beck. T0 Engineering Data, Revision 0.14. Technical Report CSD-97-931, Computer Science Division, University of California at Berkeley, 1997.

[WAK+96] John Wawrzynek, Krste Asanović, Brian Kingsbury, James Beck, David Johnson, and Nelson Morgan. SPERT-II: A Vector Microprocessor System. *IEEE Computer*, 29(3):79–86, March 1996.