Quicknet on MultiSpert: Fast Parallel Neural Network Training

Philipp Färber
*

TR-*TR-97-047*

December 1997

Abstract

The *MultiSpert* parallel system is a straight-forward extension of the *Spert* workstation accelerator, which is predominantly used in speech recognition research at ICSI. In order to deliver high performance for Artificial Neural Network training without requiring changes to the user interfaces, the exisiting Quicknet ANN library was modified to run on MultiSpert.

In this report, we present the algorithms used in the parallelization of the Quicknet code and analyse their communication and computation requirements. The resulting performance model yields a better understanding of system speed-ups and potential bottlenecks. Experimental results from actual training runs validate the model and demonstrate the achieved performance levels.

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

1 Overview

Although the Spert-II workstation accelerator [WAK+96] surpasses the performance of today's workstations by factors of 3-10, the computational demands of training artificial neural networks (ANNs) for phoneme classification requires still higher levels of performance. By combining several Spert-II boards to form the *MultiSpert* parallel computer (described in [Fär97]), we were able to further reduce training time so that very large speech databases can be processed in the course of a few days, rather than weeks.

In order to make efficient use of the MultiSpert hardware with minimal changes to the user interface, only the relevant classes in the Quicknet ANN training library were extended to implement a parallel version of the back-propagation algorithm. The Quicknet database management and pattern selection code remains unchanged, but is now executed on the host computer as master, while the computation intensive network training occurs in parallel on several Spert-II boards as slaves.

Owing to the shared-bus master/slave system architecture, the resulting performance is critically determined by the application's data transfer requirements and its computation to communication ratio. In order to scale to more than twofold performance levels relative to the best single-board implementation, a careful analysis of the communication patterns and several optimizations, such as pipelining and data compression, are necessary.

Section 2 gives a short overview of the implemented back-propagation algorithm and presents two strategies for parallel execution on several Spert-II slaves. Software pipelining is used to maximize the utilization of all slaves. In order to better understand the algorithm's behavior, we develop a performance model which allows to analyze system speed-ups.

Section 3 describes several details of the actual implementation and presents timing measurements obtained on the physical hardware. The experimental results validate the presented performance model. In addition to the achieved 'raw' performance, we also look into any changes in convergence and accuracy of the parallelized version with respect to the sequential original.

The final section summarizes the results and proposes future work.

2 MultiSpert-Quicknet Algorithms and Performance Models

In addition to classes for speech database management and pattern selection, the Quicknet library provides a number of Multi-Layer Perceptron (MLP) classes, which implement the ANN forward computation and back-propagation code. In order to achieve significant speed-ups, our parallelization efforts focus on these classes, which use computationally intensive and yet very regular algorithms.

For optimal performance on the Spert-II board, we prefer to use *bunch-mode* in place of the classic *online* training, where the network weights are only updated every *bunch-size* patterns, rather than for every presentation. As discussed in section 3.5, the use of moderate bunch sizes does not adversely affect ANN convergence or accuracy.

The sequential version of the Quicknet-MLP back-propagation training code goes through the following steps for every bunch of B input patterns:

1. Forward Propagation: The B input vectors are multiplied with the input-to-hidden weight matrix, increased by the hidden layer biases, and then transformed by a nonlinear activation function, usually the sigmoid function $1/(1+e^{-x})$. The resulting hidden unit activations are then propagated in a similar way through the hidden-to-output layer to form the MLP output activations. For bunch sizes of $B > 1$ and sufficiently large weight matrices, the bulk of computation is spent in the two highly optimized matrix-matrix multiplications.

2. Error Computation: To obtain the B error vectors, the desired output target values are

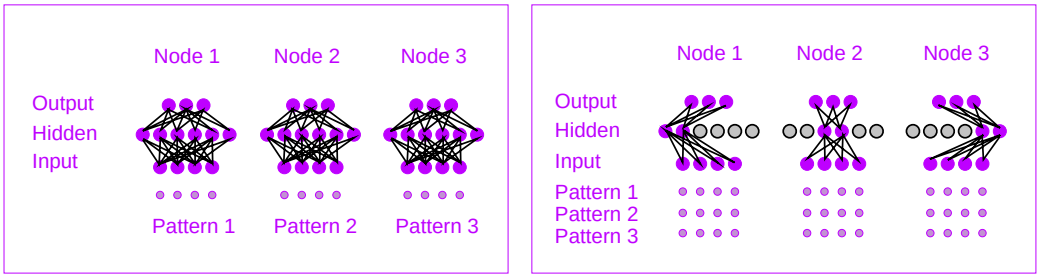

Figure 1: Pattern Parallel (a) and Network Parallel (b) MLP training algorithms

subtracted from the output activations and scaled by the derivative of the inverse activation function.

3. Weight Update: The error vector scaled by the learn rate is directly added to the output bias. The weight update for the hidden-to-output weights is obtained from the cross product of the (previously saved) hidden unit activations and the error vectors. Similarly, the updates for the hidden biases and for the input-to-hidden weights are calculated from the hidden unit error vectors, which are the result of multiplying the output error vectors with the hidden-to-output weight matrix. The weight update pass thus involves three matrix-matrix multiplications.

From a large number of parallelization strategies for this algorithm, we have implemented the two variants illustrated in figure 1 which match the MultiSpert characteristics with respect to execution model, granularity and communication requirements. Replicating the ANNs as in the *pattern parallel (PP)* version allows independent training on different input patterns, but requires regular communication of all weight updates. Distributing the ANN among the nodes as in the *network parallel (NP)* version trains different parts of the network concurrently and combines the partial results at the level of the output units for every presentation. The following sections present these two strategies in more detail and model their performance characteristics.

2.1 Network Parallel MLP

In the NP version of the MLP training algorithm, each slave works on a separate part of the full network, consisting of the weights associated with only a subset of the hidden units.

As in the sequential algorithm, each pattern is forward propagated (in parallel) to all output units, except that the output non-linearity is not applied. The master then reads all partial output values and sums them to one global output activation, from which the error values are obtained by applying the non-linearity and subtracting the target values. The error is then returned to all slaves which then perform their weight updates in parallel.[1]

The NP version shows the following characteristics:

- The available memory for the MLP weights scales with the number of slaves. Thus, the NP algorithm allows to train very large networks.
- Since MultiSpert emulates broadcasts through sequential writes, sending all input and output activations to all P nodes increases the amount of communication linearly with the number of slaves. In addition, the size of the partial MLP on every node decreases with $O(P)$, so that the communication to computation ratio effectively worsens with $O(P^2)$.

[1] For slow host machines, one could also return the summed output activations to the slaves directly and perform the error computation there. However, this requires a transfer of the target activations to the slaves, and it increases the total amount of computation.

- Increasing bunch size does not effect the algorithm's performance in principle, but larger bunches speed up slave computation and reduce the penalty of communication startup cost. We have found moderate bunch size values between 12 and 96 to work best[2].

2.2 Pattern Parallel MLP

In contrast to the NP parallelization scheme, the PP version trains one instance of the entire MLP on every slave but with different subsets of input patterns, accumulating the weight updates separately. After a fix number of patterns (the bunch size) has been processed, the master reads these updates, sums them and broadcasts the global weight update to the slaves.

The PP algorithm has the following characteristics:

- As each slave uses a copy of the entire network, this algorithm does not allow to train larger networks for more slaves.
- Every input and target pattern only has to be sent once, i.e., no broadcast is required as in the NP version. Also, there is no reason to return the output activations to the host, since error computation occurs on the slave. The communication cost of transferring the weight updates between master and slaves, on the other hand, increases linearly with the number of slaves.
- Since the PP version runs fully independent training runs in parallel, it theoretically achieves linear speed-up if the bunch size is increased with the number of nodes. For a constant bunch size, however, the communication to computation ratio still increases due to the decrease in computation time. Furthermore, the large number of network weights which have to be read from and written to every slave requires a substantial bunch size, which in turn compromises convergence[AG94]. The potential speed-up may thus be offset by an increased number of training epochs (see section 3.5).

2.3 Pipelining

In order to maximize the utilization of all slaves, the master should call the next slave routine immediately after the previous routine has finished. In the NP algorithm, however, calling the weight update routine requires the error vector, which first has to be computed from the forward pass outputs of all slaves.

This direct dependency can be relaxed using software pipelining: instead of computing the error vector immediately, the weight update routine is called with the error vector from the *previous* forward pass. The host then computes the current error while the slaves are active and stores the result for the next weight update.

The price for this performance optimization lies in increased memory requirements (the slaves also need to keep the previous forward pass activations in order to update the weights using the proper values) and in a slightly different order of weight updates:

	Non-Pipelined			Pipelined		
Step	call	input	weights	call	input	weights
1	forward	pat 0	W_0	forward	pat 0	W_0
2	update	err 0	$W_0 \rightarrow W_1$	forward	pat 1	W_0
3	forward	pat 1	W_1	update	err 0	$W_0 \rightarrow W_1$
4	update	err 1	$W1 \rightarrow W_2$	forward	pat 2	W_1
5	forward	pat 2	W_2	update	err 1	$W_1 \rightarrow W_2$

In the case of computing the forward pass on pattern 1, the pipelined version uses the original weights W_0 as opposed to the adjusted weights W_1. Due to the small values of weight adjustments, however, the effect of this delayed update on training accuracy or convergence is negligible.

[2]The hand-optimized library routines yield best performance for multiple-of-12 bunch sizes, as they use 12 of the 16 T0 vector registers for intermediate results.

No pipelined version was implemented for the pattern parallel algorithm, although waiting for all slaves to finish their transfer of the weight updates at the end of each bunch does represent a significant delay. The structure of the original Quicknet algorithm makes pipelining on the coarse level of large bunches difficult, however, and so this version was not further optimized.

2.4 Compressed Input Features

Since communication on MultiSpert is expensive, we would like to minimize the total amount of data exchanged between master and slaves. For the given application, each input pattern corresponds to a segment of acoustic speech typically nine 10ms frames wide, of which only the middle frame is classified. This ninefold overlap of the windows means that every input frame's data is transferred nine times, once in every position.

A large (nine-fold) reduction in communication is achieved by sending a continuous *chunk* of input frames to the slave, performing the windowing and randomization of the presentation order on the slave. Using this optimization allows more slaves to be used in parallel at high efficiency, but it disrupts the modular code structure by moving pattern selection methods from master to slave classes.

2.5 Performance Models

2.5.1 Training Pass

ANN training performance is measured in 'million connection updates per second' or MCUPS, which we define as the number of updated weights and biases divided by the time T_{trn} it took to compute the updates. The lower bound for T_{trn} is the time to finish the computation on a single Spert board, i.e. the sum of data transfer time T_d and computation time T_c. From figure XX, we see that T_{trn} for a MultiSpert system with P nodes remains constant, as long as the computation time T_c suffices to hide all other data transfers and the host computation time T_h. Beyond this limit, total execution time is dominated by the data transfer time to all nodes, so that adding more nodes will only decrease overall performance.

$$T_{trn} = \begin{cases} T_d + T_c & \text{for} (P-1) * T_d + T_h < T_c \\ P \cdot T_d & \text{otherwise} \end{cases}$$

The values for T_c, T_d and T_h depend on the parallelization strategy. While the NP algorithm requires all B patterns to be broadcast to all nodes, the PP version distributes the patterns and target values among the nodes. In addition, the different nodes interact in the NP version for every pattern, as the output activations are read and summed and the errors redistributed. The PP version requires interaction only once per bunch, when the $N_{wts} = N_h(N_i + N_o)$ weight updates of all nodes are gathered, summed, and the new weights rebroadcast. This yields the following expressions for T_d:

$$T_d = \begin{cases} B\frac{N_i+2N_o}{BW} & \text{for NP} \\ \frac{B}{P}\frac{N_i+N_o}{BW} + \frac{2N_{wts}}{BW} & \text{for PP} \end{cases}$$

The host computation time T_h is the most difficult to model, because it is influenced by processor speed, disk performance and any operating system overheads. We have found T_h to be dominated by the fix point conversion routines, so that T_h increases linearly with the bunch size $T_h = c_h \cdot B$ with c_h determined experimentally.

Finally, the time to perform one training pass on the Spert board depends on the network size, which we parameterize with the number of hidden units N_h, and the bunch size B. The computation time T_c' for one pattern is proportional to the number N_h' of hidden units for this node, so that we can approximate $T_c' = c_1 N_h' + c_2$), where both constants may be determined experimentally from

single Spert board runs. [3] For the NP version, each node only holds $N'_h = N + h/P$ hidden units, but has to process all B patterns for each step. For the PP version, each node processes the full network, but only B/P times.

$$T_c = \begin{cases} B * (c_1 \frac{N_h}{P} + c_2) & \text{for NP} \\ \frac{B}{P} * (c_1 N_h + c_2) & \text{for PP} \end{cases}$$

The overall training performance in MCUPS is derived by dividing the total number of updated connections by the training time:

$$perf = \frac{B(N_{wts}}{T_{trn}}$$

2.5.2 Forward Pass

The performance model for the forward pass can be derived in a very similar way:

$$T_{fwd} = \begin{cases} T_d + T_c & \text{for} (P-1) * T_d + T_h < T_c \\ P \cdot T_d & \text{otherwise} \end{cases}$$

$$\text{with} \begin{cases} T_d = B\frac{N_i+N_o}{BW}, T_c = B * (c_1 \frac{N_h}{P} + c_2) & \text{for NP} \\ T_d = \frac{B}{P}\frac{N_i+N_o}{BW}, T_c = \frac{B}{P} * (c_1 N_h + c_2) & \text{for PP} \end{cases}$$

While the NP-algorithm requires all patterns to be broadcast to all nodes, the PP-algorithm only sends every pattern once. Also, it is no longer necessary to read and write entire weight matrix updates as in the training pass.

3 Experiments and Characterization

3.1 Implementation Details

The described algorithms were implemented in C++ and compiled for a SUN workstation master and for the Spert-II slaves. In order to be able to use more than two Spert-II boards, we have used commercial SBus expander boxes, which allow to attach up to 4 boards in every slot[4].

The new MLP classes make use of the SPC library to transfer call arguments to, synchronize execution with, and read resulting data from the Spert-II slaves (see [Fär97] and the SPC man-page for details on these mechanisms). Figure 2 illustrates the execution of the inner loop of the NP algorithm for one, two and three slaves. In the latter case, performance is limited by the sequential host execution, so that the slaves can no longer be kept busy all the time. Note that there is no overlap between communication and computation on the slaves, and that broadcast has to be implemented as individual send operations. The master uses pipelining to perform the error computation when all slaves are busy.

In the PP version, a large bunch of patterns (typically over 1000) is sub-divided into smaller blocks to account for the limited buffer memory on the slaves. In this case, pipelining between bunches has not been implemented, so that the sequential reading, accumulation and broadcasting of the weight updates at the end of each bunch represents a significant performance bottleneck.

3.2 Timing Measurements

Exact execution times for the NP version have been obtained from instrumenting the code running on the master with calls to the IPM timing library. For the numbers shown in table 1, a Sparc-2

[3] we assume a sufficiently large bunch size, so that c_1 and c_2 are independent of B).

[4] Due to faulty hardware, we had to revert to only using two boards per slot

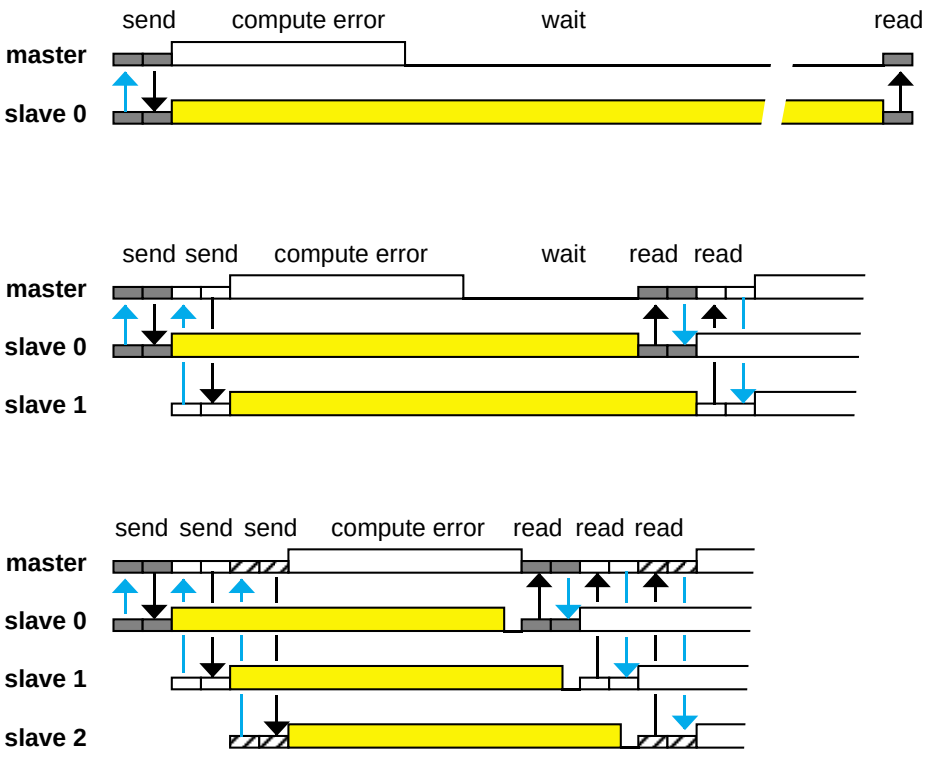

Figure 2: NP inner loop execution with 1, 2 and 3 slaves

	bunch size 48, MLP size 153:1000:56	
program block	time	comment
loop start	0	one slave (111 MCUPS)
slave.call(train,error0,input2)	2.7 ms	write data at 4.0 MB/s
local.comp_err(output1)	22 ms	
slave.synch	64 ms	wait for slave (raw: 116 MCUPS)
slave.read(output2)	2.0 ms	read data at 2.7 MB/s
loop start	0	2 slaves, pipelined (205 MCUPS)
slave0.synch	17 ms	wait for slave0 (raw: 115 MCUPS)
slave0.read(output2)	2.0 ms	read data @ 2.7 MB/s
slave0.call(train,error0,input2)	2.7 ms	write data @ 4.0 MB/s
slave1.synch	0 ms	slave1 already finished
slave1.read(output2)	2.0 ms	read data @ 2.7 MB/s
slave1.call(train,error0,input2)	2.7 ms	write data @ 4.0 MB/s
local.comp_err(output1)	22 ms	

Table 1: Timing for the inner loop of the NP algorithm

was used as MultiSpert host. We used the compressed input data optimization described in section 2.4, sending 10,000 patterns to each node before starting the training. Thus, each training step requires 56 error values (4 Bytes) and 56 partial output activations (2 Bytes) per pattern to be send to (respectively read from) each node.

While the raw slave performance remains constant at about 115 MCUPS, the overall performance does not increases linearly, due to the increasing communication overhead and the reduced amount of computation per node. In this example, the relative overall performances for 1 to 4 nodes are: 97%, 178%, 235%, and 205% of single board raw performance. The additional communication steps result in four slaves yielding lower performance than three. This behavior improves with faster host workstations, faster communication or larger network size.

3.3 Training Performance

The graphs in figure 3 shows model predictions and measurements of training performance for the two strategies. Overall, the Multi-Spert system behavior coincides well with the models. As the number of hidden units grows, scalability improves because computation increases relative to communication. For the PP version, larger bunch sizes also increase performance, as can be seen from the two top curves which are for 2000 hidden units with bunch sizes of 5000 and 10000. For a given network size, performance scales linearly with the number of slaves until execution time is dominated by communication. Beyond this point, performance drops rapidly for NP (due to the $O(N^2)$ increase in communication to computation ratio), whereas there is a more gradual decline in the PP version. Thus, PP is the more efficient strategy for smaller networks. As noted before, performance and scalability improve with a faster host and higher communication performance. The graph in figure 4 shows the modeled and measured performance of the Multi-Spert NP algorithm using a SUN Ultra-1 workstation as host. In the case of 1000 hidden units, the performance values for two and three slaves slightly exceed those shown in figure 3, and scalability increases to four nodes.

3.4 Forward Pass Performance

Figure 5 shows the forward pass performance, which is mostly communication bandwidth limited. The NP strategy has the disadvantage that all patterns are sent to all nodes, which means poor efficiency on small networks. The PP version scales much better, since every pattern is sent only

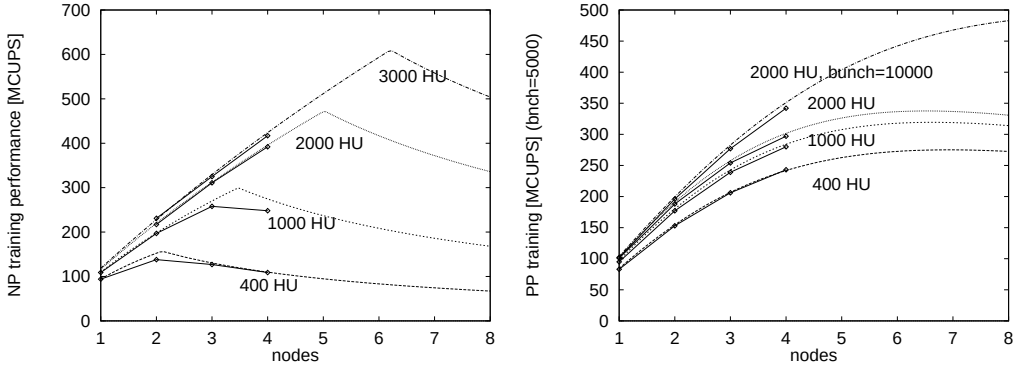

Figure 3: NP and PP training performance

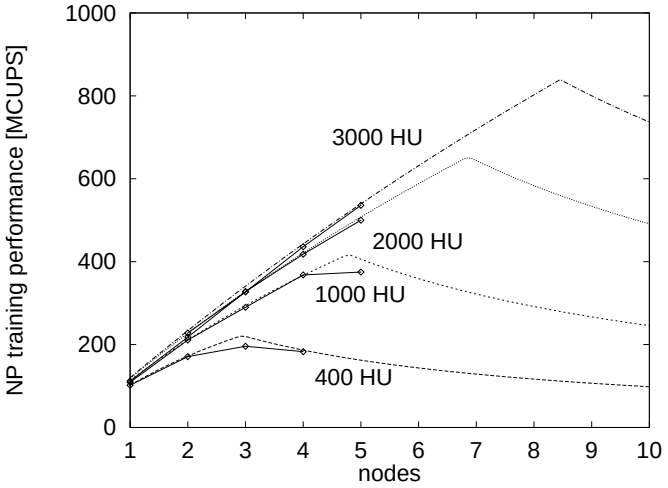

Figure 4: Fast NP performance

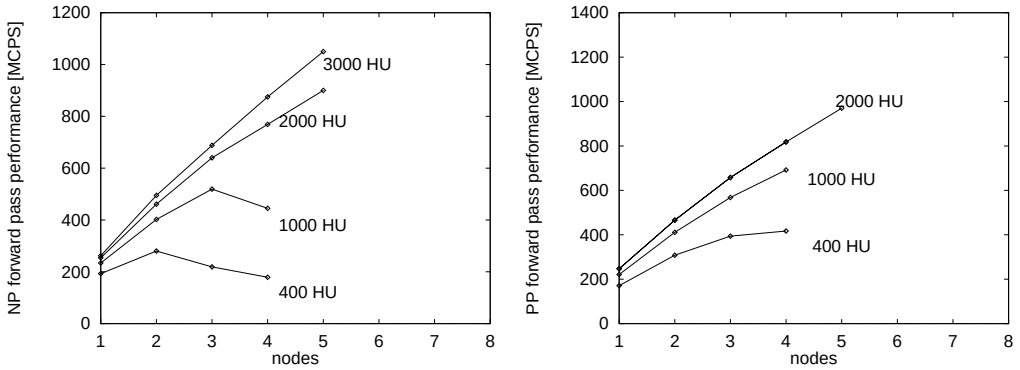

Figure 5: NP and PP forward performance

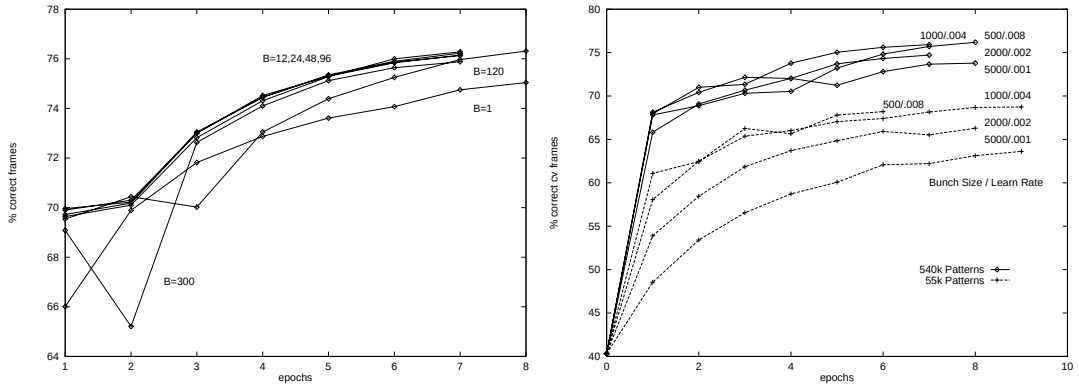

Figure 6: MLP convergence for different bunch sizes

once regardless of the number of slaves. The one advantage of the NP approach is that it can very large networks which will not fit on a single Spert-II board. With five nodes, we achieve about 1 GCPS.

3.5 Accuracy and Convergence

Although the parallelized Quicknet training performs the same computations as the sequential original, the resulting weight files are not identical. Small numerical differences are caused by a change in the order of additions or by the fact that the output non-linearity is computed on the host using floating point while the single Spert original always uses fix-point routines.

However, although the trained networks differ in their weights, we have found their classification performance equivalent as indicated by the achieved percentage of correctly classified frames. Also, there appears to be no penalty in convergence speed for the parallel algorithms, as the number of required epochs to obtain a maximum cross validation set classification remains about the same.

Experimenting with the use of very large bunch sizes as required by the PP algorithm, we have run into the problem of saturation for the fix-point weight updates. While the updates for small bunch sizes remain in the range of the actual weights, accumulating thousands of updates potentially leads to much larger numbers. Changing the fix-point exponent to accommodate this larger range reduces the precision for smaller values, which would be detrimental in later stages of the training. A good solution to this problem would be an adaptive fix-point format to always give maximal precision. In the presented results, however, we have side-stepped this issue by reducing the learning rate, thereby decreasing the magnitude of the weight updates at the cost of a slower convergence.

Figure 6 shows the effects of using bunch mode training on the convergence of the network, measured by the percentage of correctly classified frames in the cross validation set. While the differences in the left graph (for moderate bunch sizes) are negligibly small, the right graph shows a deterioration of classification performance for very large bunch sizes (combined with a reduced learn rate). As the number of training patterns and consequently the number of weight updates per epoch increases, the differences become less pronounced.

While the classification rate is a straight-forward way to characterize network performance, the ultimate goal in ASR is *word recognition*, which does not depend on correct frame classification alone. In the recognition process, not only the class with the maximum output is considered, but it is important that all outputs reflect the probabilities of the corresponding phones. Table 2 shows the results of using the trained networks in the considered ASR system. So far, there seems to be no significant loss in recognition performance from using the MultiSpert Quicknet algorithms.

network	156:400:56	156:1000:56
sequential Quicknet	7.6 %	6.7 %
NP version (4 slaves, B=12)	7.8 %	6.8 %
PP version (4 slaves, B=5000)		8.4 %

Table 2: Word error rates on numbers95 corpus

4 Summary

This technical report describes the MultiSpert Quicknet neural network training algorithm and its performance characteristics. Two variations of the algorithm with different parallelization strategies were used, the *Network Parallel* and the *Pattern Parallel* versions.

Both versions show good speed-ups in real application performance, but different parameter settings are necessary to ensure a low communication to computation ratio and thus high performance for multiple nodes. While the NP version benefits from using large networks, the PP algorithm performs best for large bunch sizes.

Looking at the convergence rate and the accuracy of the training result, we found that moderately large bunch sizes of up to 200 have no negative effects. Very large bunches with over 5000 patterns only guarantee similar results when enough training data is used, so that weights are updated sufficiently often during an epoch.

In terms of application performance, MultiSpert has well achieved our goals, and a five-board system is currently being used for MLP training at over 500 MCUPS, with a forward pass performance exceeding 1 GCPS. This means an improvement of 8 times over the latest single-board version, and of over 25 times over our fastest Ultra-Sparc workstation.

References

[AG94] David Anguita and Benedict A. Gomes. Mbp on t0: mixing floating- and fix-point formats in bp learning. Technical Report TR-94-038, International Computer Science Institute, August 1994.

[Fär97] Philipp Färber. Parallel Computing on MultiSpert. Technical Report TR-97-046, International Computer Science Institute, December 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.