Parallel Computing 19 (1993) 1393–1412
North-Holland

PARCO 802

The GF11 parallel computer

Manoj Kumar *, Y. Baransky and M. Denneau

IBM Research Divisions, T.J. Watson Research Center, Yorktown Heights, NY 10598, USA

Received 5 May 1991
Revised 29 August 1991, 25 November 1992

Abstract

GF11 is a parallel computer operational at IBM's T.J. Watson Research Center. It is based on the SIMD (Single Instruction Multiple Data) model of parallel computing. GF11 attains its peak execution rate of 11.3 GigaFlops by using 566 identical processing elements, each capable of delivering 20 MegaFlops. Each processor has its own 64 Kb static RAM that can access a 32-bit word on each floating point operation, a 2 Mb dynamic RAM that operates at one fourth of the SRAM speed, and a 1 Kb register file that provides four accesses per floating point operation. The processors communicate through a 576×576 Benes network, organized as three stages of 24×24 crossbar switches.

The network provides 11.3 Gb/sec of communication bandwidth to the processors and allows the processors to dynamically reconfigure themselves into arrays of various dimensions and sizes or other interesting interconnection patterns such as a tree, hypercube, etc. This reconfiguration can take place on every word transfer without sacrificing the bandwidth. GF11 has several architectural enhancements to circumvent the limitations of the standard SIMD model such as the ability to perform multiple operations in every instruction and the ability to modify the operations occurring within individual processors based on processor specific data.

Preliminary benchmarking efforts on some applications indicate that near peak performance can be sustained on most applications, including some that were previously believed to be ill suited for SIMD machines. Minimal restructuring of programs and algorithms is required for achieving this performance. The architecture of GF11 is summarized in this paper and the implementations of Finite Element analysis, LU decomposition, Gaussian Elimination, and Fast Fourier Transform are discussed to illustrate GF11's ability to deliver good performance with minor program restructuring.

Keywords. GF11 multiprocessor computer, architecture; Programming model; FEM analysis; LU decomposition; Gaussian elimination; FFT, performance results

1. Introduction

GF11 is a parallel computer based on the Single Instruction Multiple Data (SIMD) model of computing [10]. It uses 566 processors that receive an identical stream of instructions from a central controller. The processors communicate with each other through a three-stage Benes network [3], which operates synchronously with the processors. The synchronous operation of the network with the processors, an architectural feature unique to GF11, is an important factor in GF11's ability to sustain high performance on a broad range of applications. The network provides a unidirectional communication path from each processor to a

* *Corresponding author.* Email: mkumar@watson.ibm.com

disjoint set of destination processors. The Benes network lets the processors configure themselves logically as meshes of different dimensions and sizes, or as trees, hypercubes, etc. The processors can change their configuration on every word transfer without sacrificing the network bandwidth.

In comparison to the more popular MIMD (Multiple Instructions Multiple Data) machines [7,10,12,13], SIMD machines have simpler designs and lower manufacturing costs because the individual processors do not need the instruction memory and the instruction fetch and decode logic, which is centralized in a single place. Furthermore, in SIMD machines, the interprocessor communication can be synchronized with the execution of instructions in the processors, and therefore, synchronization overheads can be avoided. The interprocessor communication can also be scheduled a priori to avoid interference in the network, and therefore higher interprocessor communication bandwidth can be sustained.

Because of these advantages, SIMD machines were the favorites during the early days of parallel processing [1,2,18]. However, a widespread belief that SIMD machines are unsuitable for a large number of problems, caused the SIMD machine to fall out of favor over the last few years. This belief has been refuted by the recent work of Fox [11] and the users of CM-2 [17,20]. ICL's DAP [9] and CM-2 [20] are two commercially available SIMD machines.

GF11 was designed at IBM's T.J. Watson Research Center and is described in detail in [4,5,15]. The main application targeted for GF11 was the numerical verification of the predictions of Quantum Chromodynamics (QCD), a theory of particles that participate in nuclear interactions [21]. However, the GF11 design addressed the need of a wide variety of applications and as a result good performance is sustained on many applications. Some of these design features are the wide instruction word which specifies many concurrent operations per instruction (the super-scalar approach), the novel approach for interconnection network that provides the flexibility and high-bandwidth for interprocessor communication, the 256 word register file, and the handling of condition codes. These architectural enhancements also eliminate or reduce the effort required to modify and restructure the application program. Normally, programs are modified to circumvent the limitations of interprocessor communication and memory bandwidth, but in GF11 the problem does not arise for most applications and is less serious for others because of the balanced design.

GF11 is fully operational and benchmarking results show that while it can deliver sustained performance close to its peak performance on applications which are believed to be well suited for SIMD processing, it also delivers very high performance on applications which are widely believed to be ill suited for SIMD and distributed memory machines. Applications in the latter category include finite element methods on unstructured grids and molecular dynamics. In this paper we discuss the implementation of applications from this latter category on GF11. GF11 implementation of applications from the former class is presented in detail in [15], and only reviewed here. The architecture of GF11 has been described in considerable detail in [4,5,15], and is also only reviewed here for completeness.

In the next section we will briefly review the GF11 machine organization and hardware, and highlight the architectural elements we believe were essential for sustaining good performance, even on applications not considered suitable for SIMD computers previously. In Section 3 we use a simple 2-dimensional relaxation problem as an example to illustrate a programming style which is quite easy and at the same time effective in exploiting parallelism on GF11. The implementation of Finite Element analysis, LU decomposition, Gaussian Elimination, and Fast Fourier Transform codes on GF11 is discussed in Section 4, and the performance sustained on these programs is reported. The new possibilities created for making more powerful and versatile SIMD machines are discussed in Section 5. Finally, in Section 6 we give some concluding comments on the suitability of GF11 for a larger class of problems.

2. Overview of GF11 machine organization and hardware

The GF11 hardware consists of 566 identical processors connected through a 576×576 Benes network, a central controller, and 10 disk drives also connected to the network. Each processor has a peak performance of 20 MegaFlops, and therefore, the peak performance of the whole system is 11.3 GigaFlops. A high level overview of GF11 is shown in *Fig. 1*, and details can found in [4,15]. Each processor has its own data memory, and there is no shared data memory. A single copy of the program exists in the program memory in the central controller, and instructions broadcasted from the program memory are executed by all active processors as soon as they are received at the processors. The whole system operates on a 50 nanosecond machine cycle, and the network transports 1 Byte of data from every input in each cycle. Of the 566 processors, 512 are intended to be in use by a user program at any given time, and the remaining 54 function as hot spares.

The design of a GF11 processor is shown in *Fig. 2*. Each processor performs 20 million arithmetic operations per second, The key component in a processor is a 256 32-bit word register file. The register file performs four accesses every 50 nanosecs. Two accesses are for reading operands for an arithmetic operation, one access is for storing back the result of an arithmetic operation, and the fourth is for either transferring a word from the register file to the SRAM or for receiving a word into the register file from either the switch, the SRAM, or a delay pipe in the processor. The delay pipe provides an efficient way of handling boundary conditions in many algorithms.

Two operands are read from the register file in each 50 nanosec cycle and are delivered to either an integer ALU or to one of the four floating point ALUs. The floating point ALUs are pipelined and operate on a 200 nanosec clock. Staggering four of them produces a floating point result every 50 nanosec. (Faster components were not available during GF11's design period.) Two floating point ALUs perform only multiplications, and the other two perform the remaining floating point operations.

Each arithmetic operation generates a set of condition codes ($<$, $\geq$, $\neq$, ... etc.), any one of which can be selected and stored in a 7 entry condition code register. The condition code

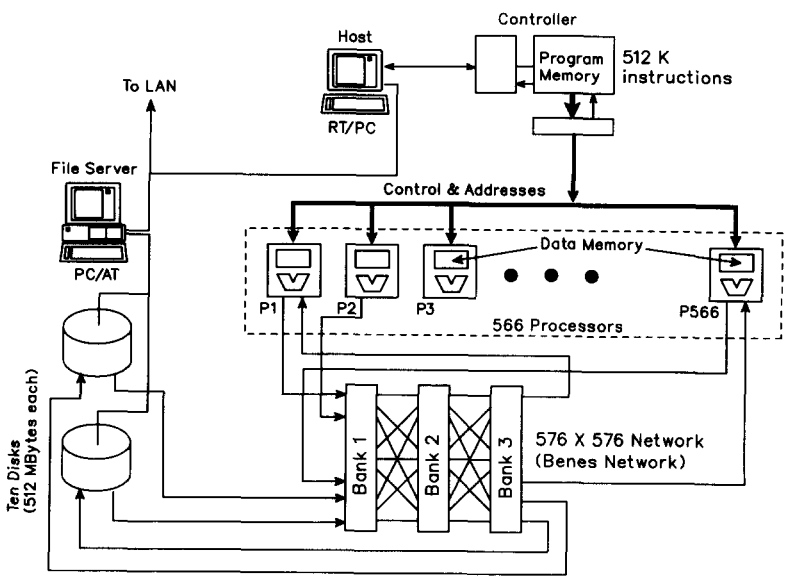


Fig. 1. Overview of the GF11 machine organization and hardware.

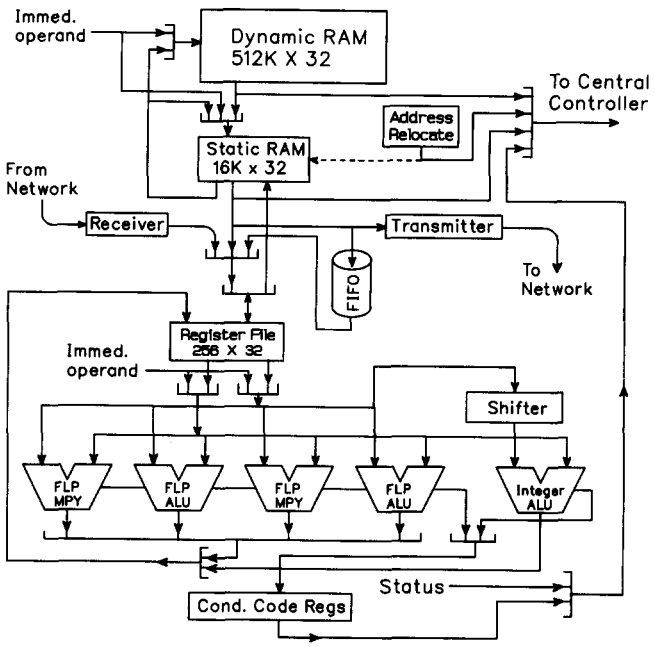


Fig. 2. Overview of a GF11 processor.

registers on each processor can be used to modify the operations being performed by that processor. Five condition code registers are accessed in each instruction. Two of them are used to abort the SRAM and the DRAM stores. The third converts a binary integer operation *A op B* to either *pass A* or *pass B*, depending on a mode bit in the instruction. The fourth condition code enables the processor to write back to the central controller, and the fifth one instructs the register file to select the data coming from the delay pipe in place of the data coming from the switch. Therefore, even though all boards in the system receive the same instruction, different boards could be doing different things depending on their local condition codes.

The communication network in GF11 is a three stage Benes network, constructed from 24×24 switches with 24 switches in each stage. This network has the following characteristics:

- *Full Connectivity:* Benes networks can provide connections from all the network inputs to the network outputs simultaneously, according to any specified permutation. Furthermore, the added capability of the GF11 switches to connect an input to multiple outputs allows the network to handle multiple broadcasts efficiently. Two passes are required over the network to allow each of several network inputs to broadcast their data to multiple outputs, provided each output is connected to a unique input [14].
- *Fixed latency:* In a SIMD parallel processing environment the non-blocking property implies that if in a given cycle all processors send a word of data to each other, they will receive their data simultaneously. The delay depends only on the hardware implementation of the Benes network, and is independent of the pattern of communication. This allows for efficient compile time overlap and synchronization of calculations and communication (just in time delivery, no buffering) at the instruction level.
- *Low Latency:* This network design approach also allows the network latency to be kept as low as the processor latency, eliminating the need to restructure the programs to mask the communication latencies.

Organization of the GF11 memory subsystem is also shown in *Fig. 2*. Each processor in

GF11 has 2 Mb of memory, and therefore, a 512 processor system has 1 Gb of memory. Memory on each processor is organized as 2 banks of 256K 32-bit words, and implemented in DRAM technology. If there are no bank conflicts, a sequence of loads or stores can be performed once every four cycles (an arithmetic operation is done once per cycle). Bank conflicts and alternating between loads and stores further reduces the bandwidth to DRAM. To prevent the DRAM bandwidth from affecting the performance of GF11, data from the DRAM is staged into a 16K word buffer implemented in Static RAM (SRAM) technology, and into a 256 word register file from there. The static RAM buffer can be accessed once every cycle.

The memory subsystem contains hardware to support address calculations for data in SRAM and DRAM. Each SRAM address can be modified by adding to it an offset from one of the 256 relocation registers. The DRAM address can be modified similarly by adding the offset from the DRAM relocation register to the DRAM address being broadcast in the instruction. A single relocation register suffices for the DRAM because the DRAM is accessed at most once every four cycles and the relocation values can be obtained from the SRAM.

The address relocation hardware gives the processors the ability to generate different SRAM or DRAM addresses based on processor specific data. This is useful in many situations, some of which are:

- *Table lookup:* for calculating transcendental functions using series expansion, for determining material properties when simulating a physical medium, if the property has different values for different ranges of some other variable being computed in the processors, and many similar situations.
- *Combinatorial algorithms:* for traversing a tree or a graph according to some rule which prescribes a different path in each processor.
- *Indirect addressing:* into arrays stored locally within the processors, and into large arrays distributed across data memories of all processors.

The latter situation occurs very frequently in all unstructured grid applications, and an efficient way to handle it on GF11 is explained in the discussion of PAM-CRASH implementation in this paper. The capabilities of the GF11 network are also crucial in handling indirect addressing across large arrays.

Though the processors in GF11 perform only one arithmetic operation in each instruction, they concurrently perform many other operations to calculate the address for and move data through the memory hierarchy. As mentioned earlier, interprocessor communication is overlapped with the arithmetic and memory operations within the processor. This contributes significantly to the ability of GF11 in sustaining close to peak performance on most applications.

3. Programming GF11

Currently programs for GF11 can be written in PL.8, a PL/1-like language. Extensions are defined in PL.8 to express parallel computations for GF11. In this section we will briefly discuss the programming model for GF11, the PL.8 programming environment used for developing, compiling and executing GF11 applications, and then we will use a simple 2-dimensional Jacobi relaxation problem to illustrate how programs are written for GF11 in extended PL.8.

3.1. The GF11 programming model

A GF11 program consists of a control program which runs on the RT/PC host, and a collection of subroutines comprising the compute intensive parts of the application which

execute on GF11. The GF11 subroutines are straight line programs (branchless) because GF11 controller does not have branching capability. Looping is done by calling the branchless subroutine repeatedly from the RT/PC.

When programming GF11, the user identifies the computation intensive sections of his application which must be delegated to GF11. The user also partitions the computational work delegated to GF11 among the available processors, taking care that all processors get identical computations which will be performed on different data. This partitioning of the computation is done in conjunction with the allocation of data accessed in that computation to the GF11 processors so that the inter-processor communication patterns required during the computation remain few and simple. To prevent the network from becoming a bottleneck, the number of accesses to the SRAMs of remote processors has to be kept below one access per four arithmetic operations.

The extensions in PL.8 allow the user to declare variables which reside on GF11 processors. Placement of data into SRAM or DRAM is currently determined by the user. The memory on each processor gets allocated in an identical manner. The permutation/broadcast patterns, which must be used by the processors to access data from other processor's SRAMs, are defined by setting the permutation tables declared in the code generator. Each permutation table defines one communication (permutation or broadcast) pattern. The ith entry in the permutation table points to the processor from which the ith processor would receive the data in the communication pattern being defined. Calculations delegated to GF11 are also expressed in PL.8 extensions. GF11 operations (add, multiply, etc.) have the syntax of PL.8 procedure calls, and these procedures are defined in the code generator.

3.2. Writing, compiling, and executing GF11 programs

The process of generating executable code for GF11 in the PL.8 programming environment is illustrated in *Fig. 3*. Once the user has identified the compute intensive sections of the application (*Fig. 3(a)*) he produces a program comprising of the control part which executes on the host and the compute intensive tasks that will execute on GF11 (*Fig. 3(b)*).

The GF11 sections of a user program are usually enveloped in PL.8 control constructs. These PL.8 constructs use the PL.8 (RT/PC resident) variables, and serve as the macro language for writing the GF11 part of the program. The control part of the program, depending on the input value of a control variable, performs one of the following functions:

- *GF11 code generation mode*
  In this mode of execution, all the subroutines comprising the GF11 tasks are invoked exactly once. Execution of these routines on the RT/PC host in this mode causes the GF11 instructions (object code for GF11) to be generated and stored on the host. The PL.8 control constructs, used as macro definition language for expressing GF11 tasks succinctly, are executed. As a result, the procedure calls defining GF11 operations are invoked repeatedly to generate long sequences of GF11 code (*Fig. 3(c)*). For example, a PL.8 loop around a set of GF11 operations will cause the GF11 code for this set of operations to be generated repeatedly, which is the standard technique for unrolling GF11 loops. This is explained in some detail in Section 3.3.
- *Execution mode*
  In this mode, the GF11 object code generated by the earlier mode is loaded to the GF11 instruction memory, and the control program on the host now invokes the GF11 tasks to carry out the desired calculations.
- *Simulation mode*
  This mode is used before the code generation and execution modes described earlier to facilitate the development of GF11 applications. When a GF11 task is invoked in this

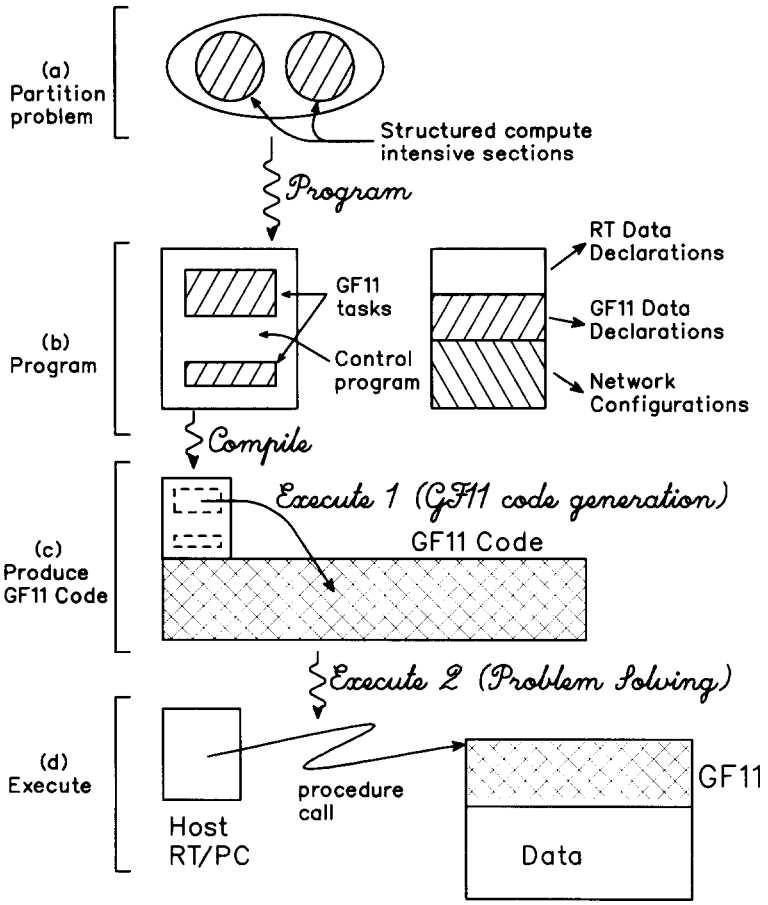


Fig. 3. Process of writing, compilating, and executing GF11 programs.

mode, it gets interpreted to simulate its execution, rather than being compiled to produce GF11 code. The facility to interpret/simulate GF11 tasks is part of the code generator.

3.3. The Jacobi relaxation example

In the Jacobi relaxation example we compute a $N \times N$ matrix A, in which each element is the average of the four neighbors of the corresponding element in another $N \times N$ matrix B ($A_{i,j} = \frac{1}{4}(B_{i,j-1} + B_{i-1,j} + B_{i,j+1} + B_{i+1,j})$). This step is executed repeatedly, interchanging the roles of A and B. We will assume periodic boundary conditions and use N processors to solve the problem.

In this example we have only one computation intensive section which represents the entirety of the code. Real programs however, have several such sections in addition to the control constructs and some sequential code which is executed on the host. We generate a sequence of instructions which, when executed once on GF11, will carry out one iteration of Jacobi relaxation. Each processor will contain one row of the A and B matrices and compute the new values for the elements in those rows. When a processor i is computing the new value for an element B_{ij} ($0 \leq j < N$), two of the four neighbors of $A_{i,j}$, $A_{i-1,j}$ and $A_{i+1,j}$, reside on other processors, while $A_{i,j-1}$ and $A_{i,j+1}$ are available locally. Therefore, we require two

```
(1)   DeclareArrays : Proc;
(2)     DclA( a_sr, scSR, rcReal, mic(LEN), ecFloat);
(3)     DclA( b_sr, scSR, rcReal, mic(LEN), ecFloat);
(4)     DclA(aZ_sr, scSR, rcVirt, mic(LEN), ecFloat);
(5)     DclA(bZ_sr, scSR, rcVirt, mic(LEN), ecFloat);
      End Proc DeclareArrays;

(6)   DeclareRelocates : Proc;
(7)     DclCR(craZ_sr, crSR);  Anchor(aZ_sr , craZ_sr);
(8)     DclCR(crbZ_sr, crSR);  Anchor(bZ_sr , crbZ_sr);
      End Proc DeclareRelocates;

(9)   DclSwitch : Proc;
(10)  Dcl Pr Integer;
(11)    Do Pr = 0 To PrMax-1;
(12)      SwPerms(0,Pr) = MOD(Pr-1,PrMax);
(13)      SwPerms(1,Pr) = MOD(Pr+1,PrMax);
        End Do Pr;
      End Proc;


(14)  gmc_SWEEP : Proc;
(15)  DCL (j,jp,jm,r(4,LEN)) Integer;
(16)    Do j = 1 To LEN;
(17)      jp = MOD(j      ,LEN) + 1;
(18)      jm = MOD(j-2+LEN,LEN) + 1;

(19)      SwLd(aZ_sr(j),r(1,j),swa(0));
(20)      SwLd(aZ_sr(j),r(2,j),swa(1));
(21)      r(2,j) = FlAdd(r(1,j),r(2,j));

(22)      Ld(aZ_sr(jp),r(3,j));
(23)      Ld(aZ_sr(jm),r(4,j));
(24)      r(4,j) = FlAdd(r(3,j),r(4,j));

(25)      r(2,j) = FlAdd(r(2,j),r(4,j));
(26)      r(2,j) = FlMul(FlConst(0.25),r(2,j));
(27)      St(r(2,j),bZ_sr(j));
        End Do j;
      End Proc;


(28)  Run_SWEEP : Proc(aZ_sr , bZ_sr);
(29)  DCL (aZ_sr,bZ_sr) pgElt;
(30)    Point(craZ_sr , aZ_sr);
(31)    Point(crbZ_sr , bZ_sr);
(32)    If rMode = rMode_Run Then Fire(MCode_SWEEP);
(33)    Else gmc_SWEEP;
      End Proc;

  /*---------------------- COMPUTATION ------------------------------*/
(34)    Do iter = 1 To itermax;
(35)      Run_SWEEP(a_sr(1),b_sr(1));
(36)      Run_SWEEP(b_sr(1),a_sr(1));
        End Do iter;
```

Fig. 4. GF11 program for the Jacobi relaxation problem.

communication patterns; one to import the value of the top neighbor $A_{i-1,j}$ and the other for the bottom neighbor $A_{i+1,j}$.

The actual program for the above problem is shown in *Fig. 4*. The first five lines declare the data structures allocated in the GF11 processors. a_sr and b_sr are one-dimensional arrays of length 'LEN' allocated in SRAM. They are the rows of the matrices A and B, and contain floating point numbers. aZ_sr and bZ_sr are virtual arrays that are not assigned any storage. Addresses for elements of virtual arrays are computed in the central controller by adding the offsets stored in the SRAM relocation registers of the central controller to the SRAM address broadcast in the instruction. By relocation, virtual arrays can be mapped to real arrays of the same size and shape. In lines 7 and 8 in *Fig. 4* the *DclCR* function allocates registers in the central controller for storing offsets, and the *Anchor* specifies that the addresses for elements of aZ_sr are always relocated using the offset stored in craZ_sr. crbZ_sr is used similarly to relocate bZ_sr.

The code for updating one row of the matrix is shown in lines 16 through 27. It uses the virtual arrays aZ_sr and bZ_sr as input and output matrices, so that the same code can be used to take the real matrices a_sr and b_sr as inputs in alternate iterations to produce b_sr and a_sr respectively as result matrices. The controller relocation registers are modified to achieve this in lines 30 and 31. When statement 35 makes the call to Run_Sweep, craZ_sr is adjusted to map virtual array aZ_sr to real array a_sr, and crbZ_sr is adjusted to map virtual array bZ_sr to real array b_sr. However, when line 36 makes the Run_Sweep call, aZ_sr is mapped on b_sr and bZ_sr is mapped on a_sr.

The GF11 program written in PL.8 extensions to update one element within a row is in lines 19 through 27. This code is enveloped in a PL.8 'Do' loop at line 16 whose index variable j is an RT/PC variable. The loop also calculates two RT/PC variables *jp* and *jm*, which are used in GF11 calculations in lines 22 and 23. The entire program is run in one of the two modes indicated by the system variable rMode in line 32. When the value of rMode is not equal to rMode_run, the gmc_SWEEP subroutine is invoked, and the execution of the code generator subroutines comprising the GF11 program causes the GF11 program to be compiled and GF11 code is generated. In each iteration of the do loop in line 16, new GF11 code is generated for processing the next element of the row, and the loop required to index over the elements of the row is completely unrolled. No calculations are performed on GF11 in this mode. Once the GF11 code has been compiled in the above manner, executing the entire program with rMode equal to rMode_Run causes the actual calculations to be performed on GF11 using the earlier generated code.

Lines 9 through 13 in *Fig. 4* define the interprocessor communication patterns required. A_{Prj} is stored in processor Pr as a_sr_j. Its top neighbor $A_{Pr-1,j}$ and its bottom neighbor $A_{Pr+1,j}$ are stored in processors $Pr-1$ and $Pr+1$ respectively in the same location a_sr_j. Neighbors of B are stored similarly.

In extended PL.8, all variables which are not assigned storage in either the SRAM or the DRAM (such as the r(,,) variables in the Jacobi code), are automatically allocated by the compiler into registers. The do loop in line 16 provides indices to process all elements of a row. Statements 19 and 20 show the top and bottom neighbors of an element being loaded using the network. Lines 21 through 27 show the remaining loads, the averaging of the neighbors, and the storing back of the results.

4. Applications

Several algorithms have been implemented on GF11 to understand its suitability for Scientific/Engineering applications. In this section we have chosen some of these algorithms

to discuss the implementation issues, the performance sustained, the architectural features of GF11 which allow good sustained performance where other machines with distributed memory have difficulty, and the reasons for the sustained performance being less than the peak performance. Structural analysis was chosen because it is widely believed to be ill suited for SIMD machines because of the indirect addressing involved. The matrix applications and FFT were chosen because they are simple, widely known, and bring out the capabilities/weaknesses of the architecture.

4.1. Structural analysis application (Pam-Crash)

Pam-Crash is a structural analysis code developed by ESI France [24]. It is used for analyzing dynamic response of structures. It is used primarily to study automobile crash worthiness. This is a finite element code. Time integration is done by an explicit finite difference scheme to compute the acceleration, velocity, and displacement of the discretized points in the structure as a function of time.

We used the QFORCEI subroutine of the Pam-Crash code. This routine computes the intermediate results needed for calculating the forces acting on each element. The QFORCEI subroutine was chosen because the calculations in this routine are representative of the entire computationally intensive part of the code. This routine has five types of computational loops, and two distinct communication patterns. The communication patterns are analogous to the gather step in vector processing. They arise frequently from the need to collect the variable associated with the nodes of an element when the calculations are performed on the elements, as illustrated in *Fig. 5*.

Because of the frequent indirect addressing associated with the gather/scatter steps, finite element codes are considered to be ill suited for SIMD machines. Furthermore, since each node is involved with multiple elements, the communication from nodes to elements is not a permutation but a broadcast/multicast pattern instead. However, the indirection vector used to access the nodes corresponding to each element does not change with time. Therefore the movement of nodal values from the processors in which they are stored to the processors in which they are required for calculations can be scheduled over the network at compile time on GF11. Thus, remarkable efficiency can be achieved for the communication step.

To implement the code on GF11, we distributed the approximately 10,000 nodes and elements in the problem equally among the 500 processors without worrying about the locality of access. To generate the network permutation patterns needed to move the nodal values, a virtual Benes network is implemented in software as shown in *Fig. 6*. Each network input and output corresponds to a node and an element of the problem, and elements/nodes assigned to any processor are assigned to the same network switch. This $10{,}000 \times 10{,}000$ virtual

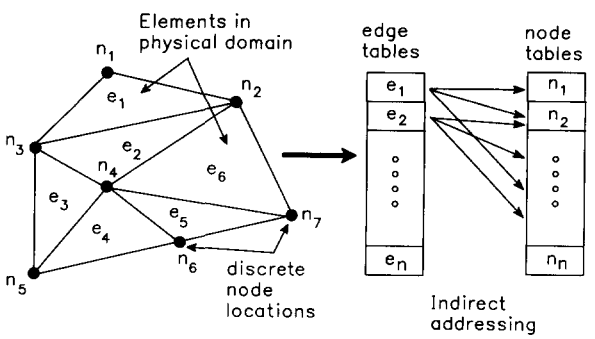


Fig. 5. Indirect memory access pattern in Pam-Crash.

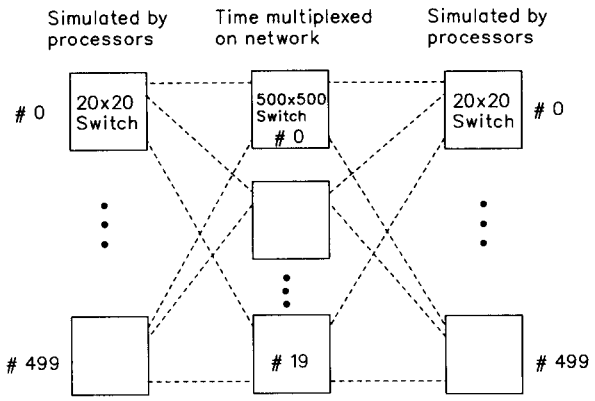


Fig. 6. 10,000 × 10,000 virtual network.

network can realize any broadcast/multicast pattern from the nodes to the elements in two passes over the network [14]. It comprises of 500 20 × 20 switches in the first and the third stages, and 20 500 × 500 switches in the middle stage. The switch settings for the virtual network are determined by using the same algorithm which is used for the real Benes network. 20 GF11 communication cycles are used to implement one pass of the virtual network. (Each communication cycle is equal to the four clock cycles required to transfer one word over the network.) In these 20 cycles each processor simulates the function of some first stage and the same third stage switch, by using the SRAM/DRAM relocation features, and the GF11 network simulates the function of the 20 middle stage switches. Thus, 2 passes of the virtual network are implemented using 40 GF11 communication cycles, or 2 communication cycles (8 processor cycles) for each value moved. 40 GF11 network communication patterns are used for the purpose.

The computation loops in the problem execute at efficiencies varying from 60% to 80%. The computation is completely overlapped with communication, and the communication step is the performance limiting step in the current implementation. We did not run the code on an actual data set, but by looking at the compiled output we infer that the computation intensive part of Pam-Crash is expected to perform at about 55% efficiency on GF11 (i.e. 5.5 Gigaflops).

It is important to note that no attempt was made for a sophisticated partitioning of the problem that improves the locality of access. Such partitioning is a necessary headache for parallel architectures which lack the high bandwidth and routing capabilities of the Benes network. We still achieved good performance, which could be improved if we optimized the data partitioning for the locality of access. Communication cost in our current implementation can also be reduced by merging a few communication steps. Communication costs also remained low on GF11 because the network replicates the nodal values being communicated when multiple elements require the same nodal value, a capability lacking in other parallel machines.

4.2. Solving dense linear systems using LU decomposition

This algorithm, often used to solve dense system of linear equations, was implemented on GF11 in two different versions to processes small and large matrices in an optimal manner. Small matrices can be completely contained in the fast SRAM memory of GF11 (2500 × 2500 matrix for 500 processors). The LU decomposition included the forward and back substitution steps required to solve a system of linear equations.

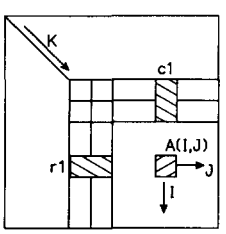
4.2.1. The SRAM version

The SRAM version of the algorithm was restructured so that the inner most loop would subtract the product of the pivot row and pivot column in the appropriate region of the coefficient matrix. With this restructuring the algorithm has a control structure with a triple nested loop. The outermost loop consists of two major steps, finding the pivot element within a specified column followed by the update of the matrix using the pivot element, pivot column, and the pivot row. The middle loop is used to index through the rows of the coefficient matrix, and the inner loop to index through the columns.

Consecutive rows were assigned to the processors evenly. Therefore the middle loop is parallelized. Though this partitioning of data simplifies the programming of the update step, it creates two difficulties. Firstly, each column is now spread across all processors and interprocessor communication is required to select a pivot. Secondly, now the pivot row has to be broadcasted to all processors. The latter is not a problem in GF11 because the network can support broadcast communication. To find the pivot element, first the local maximum is determined in each processor, and then binary reduction is used requiring log n communication steps. Because the ALU pipeline depth is 25 clock cycles and interprocessor communication latency is 20 cycles, this step takes almost 1000 clock cycles, which is almost a one processor performance. However, the pivot finding step is much smaller than the update step, and therefore the aggregate performance is not affected. The impact would diminish further when larger matrices are used.

The static RAM bandwidth will degrade the performance of GF11 by 9%, if we follow the above approach strictly. For example, when processing a 2500×2500 matrix, using 500 processors, each processor updates five elements of a column using five subtracts and five multiplies, requires five SRAM accesses for the coefficients to be updated, and five accesses to store back the updated values. The processor containing the pivot row has to do the same calculations, and in addition it has to access the pivot row element in that column from the SRAM and broadcast it to all processors, using up 11 cycles for 10 arithmetic operations. To avoid this penalty we process two columns of the matrix between the update steps, as illustrated in *Fig. 7*. Essentially, we follow the first pivot step by an update of the next column, find the second pivot for the second column, and subtract the suitable multiple of the first pivot row from the second pivot row. To update the remaining matrix, the dot product of two two-element vectors is subtracted from each element of the matrix. 20 operations are performed on the five elements accessed and stored back per column in each processor, leaving sufficient spare SRAM bandwidth to access the pivot rows. Another optimization incorporated was to overlap this update step with the two pivoting steps of the next iteration.

The SRAM version of the LU decomposition algorithm achieves 5.6 GigaFlops on GF11 using 500 processors (including the forward and back substitution steps to solve the equations). Load imbalance on processors is responsible for limiting the performance of this algorithm on GF11 to 6.7 GigaFlops. In each iteration of the outer loop, the number of rows and columns



Basic Step

```
Do 1 K = 1, 1000, 2
   Do 1 I = 1, 2
      Do 1 J = K+1, N
1        A() = A() - r1.c1
```

Fig. 7. Implementation of LU decomposition.

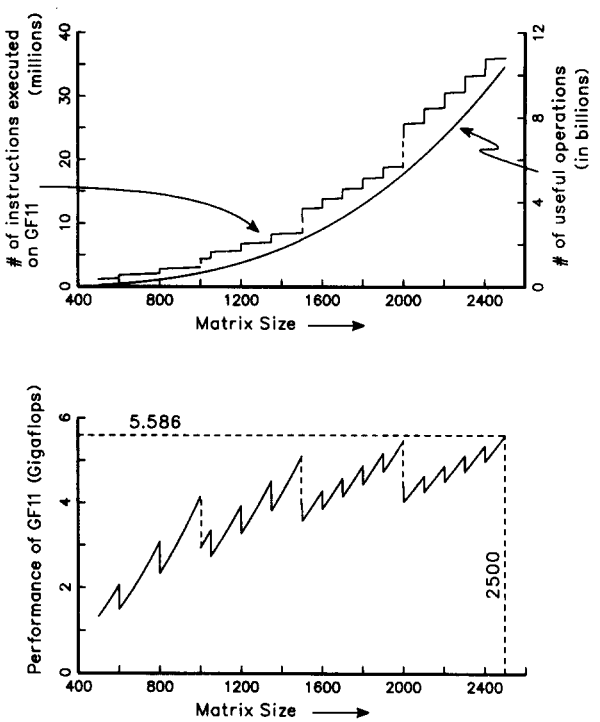


Fig. 8. Performance of LU decomposition.

being processed in the coefficient matrix decreases by one. Since partial pivoting is employed, the sequence in which the rows become inactive can not be predicted at compile time. In all iterations (except the last four) some processor has the possibility of having five active rows. The SIMD implementation of the algorithm has to accommodate this possibility, and five rows are processed conditionally in each iteration of the outer loop. However, the columns become inactive in a known order, and inactive columns are not processed. Thus, to perform the $\frac{1}{3}N^3$ multiply-add operations required for the LU decomposition, we have to issue $\frac{1}{2}N^3$ multiply-add operations. One third of the issued operations are wasted on inactive rows. Thus, the maximum performance achievable in this situation is 6.7 GigaFlops. It should be noted that this problem is generic to distributed memory machines and is not specific to SIMD machines or GF11.

The 5.6 GigaFlops achieved falls short of the 6.7 GigaFlops possible because of the inefficiencies in the pivoting step, the latency of ALU pipelines in the processors, and idling of the ALUs. The innermost loop is unrolled a 100 times to reduce the impact of ALU pipeline latency. The number of columns processed by the inner loops reduces by one in each iteration of the outer loop. Whenever the number of columns to be processed is not an exact multiple of 100, the ALUs have to idle to the next multiple of 100. Unrolling the inner loop 100 times proved optimal for us. The relatively inefficient forward and back substitution steps also degrade the performance slightly.

The performance of the above algorithm as a function of the matrix size is shown in *Fig. 8*. The number of useful operations is given by the formula $\frac{2}{3}N^3 + 2N^2$, for $N \times N$ matrix. The performance curve has the shape of a high frequency saw-tooth superimposed on a lower frequency saw tooth. The low frequency saw-tooth appears because work load is distributed unevenly among the processors unless the matrix size is a multiple of 500, and the degradation

is proportional to the number of processors that have less work than the maximally loaded processor. The high frequency saw-tooth curve exists because of the large number of columns processed in a single iteration of the inner most loop. When the matrix size is not an exact multiple of this loop unfolding count, the matrix has to be padded with zeros to the next exact multiple, resulting in wasted calculations. To get optimal performance, the inner most loop was unfolded 200 times for matrices of size less than 1000×1000, 150 times for matrices of sizes between 1000×1000 and 1500×1500, and 100 times for larger matrices.

4.2.2. The DRAM version

The LU factorization of dense matrices with column pivoting, and the solution of the resultant dense triangular system was also implemented for large matrices. DRAM must be used for matrices larger than 2500×2500, which are referred to as large matrices in this paper. Matrices of sizes up to 6000×6000 can be handled using this algorithm. We assume that the size of the matrix N is multiple of P, the number of processors.

The algorithm is partitioned and parallelized as follows. Consecutive columns of the matrix are assigned to processors in a round robin fashion as show in *Fig. 9(a)*. Throughout *Fig. 9*, the shaded region of the matrix shows the elements of the matrix being modified at the corresponding step. Therefore, each processor has N/P columns of the matrix. The advantage of this mapping is that it leads to good load balance throughout the factorization, because as rows are eliminated, each processor still has to do the same amount of work on each active column, and as each column is eliminated, the number of active columns in any

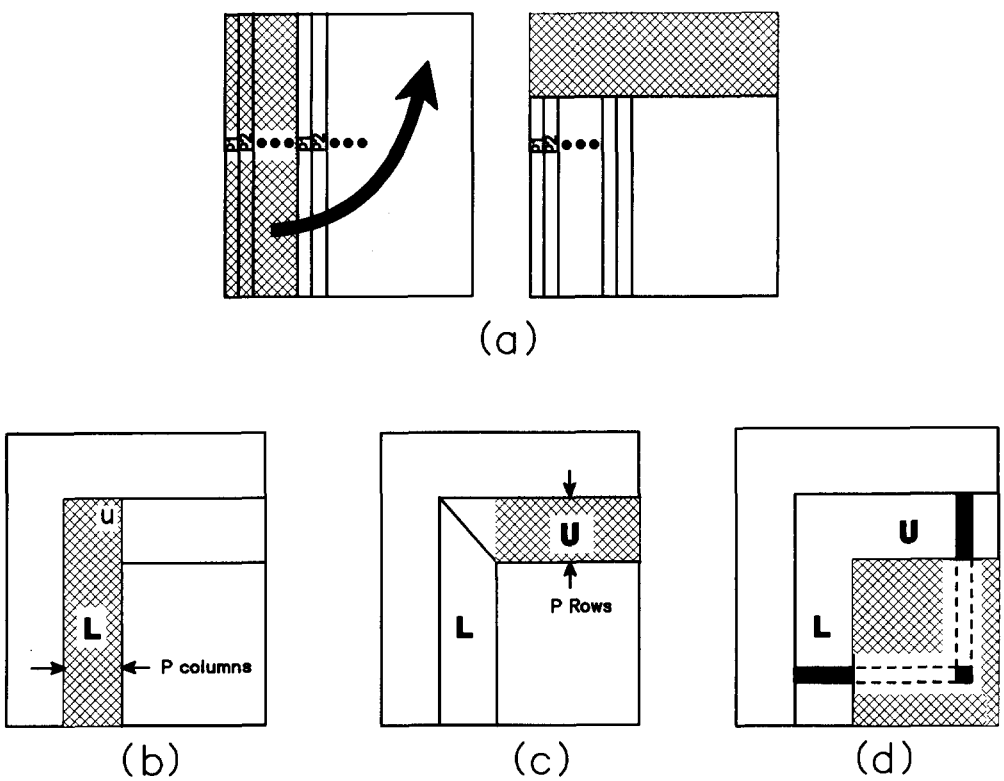


Fig. 9. LU decomposition in DRAM.

two processors can differ by at most 1. However, the algorithm must be carefully restructured to reuse the data transferred from DRAM to SRAM to avoid the DRAM bandwidth bottleneck. The strategy adopted was to perform the following rank-P update N/P times. The rank-P update proceeds as follows:

- We transpose a sub-matrix of P consecutive columns of the matrix (one column from each processor) so that now each each processor has N/P rows of the submatrix, and P columns as shown in *Fig. 9(a)*.
- This sub-matrix fits entirely in SRAM and we factor this sub-matrix using the SRAM decomposition algorithm described in the preceding section.
- Next we restore the rows of the submatrix to it original position, and permute the rows of the entire matrix to place the pivot elements generated in the previous step on the main diagonal. At this stage the P columns containing the pivot elements have the correct values for the L matrix. The $P \times P$ upper triangular matrix above the main diagonal has the correct values for the U matrix. (See *Fig. 9b*).
- Then we use the P columns of the L matrix identified in the SRAM decomposition step to update the unprocessed columns of the P pivot rows, to generate the U matrix values for the P pivot rows. (See *Fig. 9(c)*).
- Finally, we do a rank P update to the remaining unfactored piece of the matrix. This update consists of a subtracting from each (i, j) element of the unfactored matrix, the dot-product of the P newly formed columns in row i of the L matrix with the P newly formed rows in column j of the U matrix. (See *Fig. 9(d)*).

To solve a 5400×5400 system of equations on GF11, using 450 processors (12 columns per processors), this algorithm achieves 82% efficiency, yielding 7.4 GigaFlops.

4.3. Solving dense linear systems using Gaussian elimination

The Gaussian Elimination algorithm solves a system of linear equations by eliminating the variables, one at a time, from all but one equation. The sequence of transformations used in the process, if applied to a identity matrix produces the inverse of the coefficient matrix. For matrices smaller than 2500×2500, The algorithm used has a structure similar to the one used for the LU decomposition where the matrix is partitioned among the processors by rows. Column wise partial pivoting is performed in the outermost loop, and the middle and inner loops index over the rows and columns respectively.

GF11 achieves 9.3 Gigaflops when performing Gaussian Elimination on a 2500×2500 matrix, using 500 processors. Unlike LU decomposition, all rows of the coefficient matrix remain active throughout the algorithm, and thus load imbalance is not present. The inefficiency in the pivoting step, the latency of ALU pipelines in the processors, and idling of the ALUs (due to unrolling of the inner loop) account for the loss of performance.

Matrices larger then 2500×2500 are processed from the DRAM with better efficiency. The efficiency improves for large matrices because the relative overhead of finding the pivot decreases with matrix size (the overhead of accessing DRAM is negligible). To perform Gaussian Elimination on a $N \times N$ matrix, N/P consecutive rows of the matrix are assigned to each processor.

P consecutive columns of the matrix are handled at a time. These P columns are moved into the SRAM, where they are processed as described in the preceding section. However, during the processing, the multipliers used to update the rows and the row indices of the pivot elements are saved in SRAM for future use in applying identical updates to the remaining columns. The processed columns are returned to the DRAM. The pivot element values and the row indices are now used to update the remaining columns by bringing them into SRAM, P columns at a time. N/P iterations of this step yield the inverse of the coefficient matrix.

Using 500 processors, GF11 required 45.2 seconds to compute the inverse of a 6000×6000 matrix, sustaining an average execution rate of 9.5 GigaFlops.

4.4. FFT (Fast Fourier Transform)

A 2-dimensional $N \times N$ decimation in frequency FFT was implemented on GF11. The 2-dimensional FFT of $z(k_1, k_2)$ is:

$$\hat{z}_{j_1 j_2} = \sum_{k_1} \sum_{k_2} \omega^{j_1 k_1 + j_2 k_2} z_{k_1 k_2}, \quad \text{where } \omega = e^{i2\pi/N}, \quad \text{a } N\text{-th root of unity.}$$

This is accomplished by first performing a 1-dimensional FFT on the columns of z, and then a 1-dimensional FFT on the rows of the result. The algorithm is parallelized and partitioned as follows. Each processor receives N/P complete columns of z. Each processor performs a 1-dimensional FFT on all the columns it contains, with bit-reversal. This occurs without any inter-processor communication. After all the 1-dimensional FFT's have been performed, the matrix of transformed values is transposed across the machine. No computation occurs at this stage. Another complete set of FFT's are then performed and the matrix is again transposed to complete the algorithm.

In order to hide the long ALU latencies, and to relieve the pressure on memory bandwidth, the innermost loop of the 1 dimensional FFT algorithm processes only a section of the column, but the processing on this data is done for several stages of the FFT. This minimizes the amount of microcode generated. The roots of unity are precomputed and stored in auxiliary arrays as are the various pointers needed for accessing these roots and for implementing the transpose. The performance of the FFT algorithm on GF11 can not exceed 8.3 GigaFlops due to the imbalance between the add and multiply operation (6 adds and 4 multiply operations in a basic butterfly). The 1-dimensional FFT's achieved 96% of this number, or 80% of the GF11's peak performance. The bit-reversal and the transpose are not overlapped with any computation and degrade performance to 70% of peak. A 1024×1024 FFT using 512 processors achieved 7.17 GigaFlops.

4.5. Other applications

As mentioned earlier, to obtain good performance on GF11, one must be able to partition the calculation into as many identical pieces as the number of processors, and one must also partition the data in a manner such that the intercommunication pattern induced by the data partitioning does not become a performance bottleneck. In our survey of applications we rarely come across programs where the calculation cannot be partitioned properly, but partitioning of data usually requires some care.

It takes several hundred thousand steps to compute the switch setting for a given communication pattern and to load it in the switch. In all the applications discussed earlier, the communication patterns induced by data partitioning were known at compile time, so that the switch setting required to carry them out could be precomputed and preloaded in the network. If interprocessor communication patterns can be determined only at the time of program's execution, and their use is not long enough to amortize the cost of setting the switch, one has to find a more efficient way of carrying out such communication such as allowing the processors to broadcast sequentially, or simulating a packet switched network on top of the Benes network.

Molecular dynamics is one application area where the interprocessor communication is of this type [23]. Molecular dynamics simulations involve evolving the spatial configuration of atoms in molecules. To compute the new position of an atom, its interaction with all other

atoms must be computed. Since this is a N^2 process, only those pairs which are close enough are actually considered. During the course of the simulation, the set of atoms which are close to any given atom changes, changing the set of interacting atoms and hence the communication pattern. This change is data dependent and cannot be predicted in advance. In these simulations, the atoms are assigned to the processors of a parallel computer. Information about the interacting atoms must be available to processors during this computation.

In the message passing mechanism implemented on GF11, a packet is constructed for each datum to be communicated, which contains the address of the target processor. The processors are configured as a ring, and the packets are circulated between the processors, with each processor retaining a copy of all the packets addressed to it. If P is number of processors, this algorithm takes $55 \times P$ cycles (1400 microseconds for $P = 512$) to guarantee delivery of all packets. The shuffle-exchange topology would be more efficient if the number of processors exceeds 50. For 512 processors, using the shuffle-exchange topology and uniformly distributed destination addresses, an average of 5.3 (standard deviation 0.48) passes through the network are required before all packets are delivered, requiring 650 microseconds.

In addition to the applications already discussed, simulation of neural networks [22], factoring large numbers, bi-conjugate gradient method with incomplete LU preconditioner (the kernel of a large Finite Element program), Matrix Multiply, Shallow Water Equations [17], and the simulation of galactic evolution [6,16], are some of the other applications that have been implemented on GF11 and have sustained good performance. We briefly summarize the last three below. Details on these applications can be found in [15].

To multiply two $N \times N$ matrices A and B, on GF11 with P^2 processors, the matrix is partitioned into $N/P \times N/P$ blocks on the $P \times P$ grid of processors. Each processor computes a $N/P \times N/P$ block of the result matrix C. The blocks of A and B matrices are stored on the processors in a staggered manner to allow each processor to compute one product term of the result matrix block assigned to it. After computing the product term, each block of the A matrix is cyclically shifted to the processor on its right in the same row. Each block of B is similarly cyclically shifted to the processor above it in the same column. The evolving block of the result matrix C does not move. After P iterations of block multiply/accumulates and shifts, the result is available. To implement a 1024×1024 matrix multiply using 512 processors, a 32×32 grid of 1024 logical processors was used, each physical processor acted as 2 horizontally adjacent logical processors. This algorithm achieved 10 GigaFlops on GF11 for the above problem size.

NCAR is a standard benchmark for structured, explicit fluid dynamics calculations. A 512×512 grid calculation was implemented on GF11. Each processor received 1 row of data.

Table 1
Performance sustained by GF11 on scientific/engineering applications

Application	Number of processors used	Performance (GlgaFLOPS)	Problem size
Pam-Crash (finite element method)	500	5.5	10,000 elements
TPP (linear algebra, LU decomposition)	500	4.3	1000×1000
TPP (linear algebra, LU decomposition)	500	5.6	2500×2500
TPP (linear algebra, LU decomposition)	450	7.4	6000×6000
Gaussian elimination	500	9.3	2500×2500
Gaussian elimination	500	9.5	5400×5400
2-D FFT	512	7.2	1024×1024
Shallow Water equations (weather code)	512	7.5	256× 256
Matrix multiplication	512	10.0	1024×1024

Though there are less communication intensive decompositions, this decomposition is easily handled by the GF11 switch and requires minimal program restructuring. The calculation achieved 7.5 GigaFlops, which is 83% of the peak speed attainable by the algorithm on GF11, in presence of the add/multiply imbalance.

The GALAXY code simulates the evolution of galactic structures. This code is also a structured, explicit calculation on a 3-dimensional grid. However, it incorporates additional physics and chemistry to model the interaction of stars and gas of a representative piece of the galaxy for a significant length of time. Additionally, an adaptive time-stepping criterion was used to maximize the time step, requiring a global communication step. This code achieved 7 GigaFlops.

The results of the applications discussed in the previous section are summarized in *Table 1*.

5. Discussion and future directions

The hardware technology for implementing GF11 was selected in 1984. Then the floating point ALU chips were capable of delivering 5 MegaFlops, the static RAM chips could store 16 Kb of data, and the dynamic RAM chips could hold 256 Kb of data. In todays technology we have ALU chips which can deliver 100 MegaFlops, static RAM chips are twice as fast and can hold 4 Mb of data, and the dynamic RAM chips can hold 16 Mb of data. With this latest technology it it possible to make processor boards which are twenty times as powerful as the GF11 processor boards, and switch boards can be made which are proportionately faster. We are investigating the use of this latest technology to build a SIMD machine architecturally similar to GF11, which would be significantly-more powerful and compact.

The user interface in GF11 is primitive and the compilation techniques are also quite elementary. Nonetheless, the compiled code produced by the GF11 compiler is near optimal, as illustrated by the performance numbers. If the currently emerging/mature compiler technologies were employed in the GF11 compiler, the user would be freed from the burden of isolating the compute intensive sections of his program, and partitioning these calculations among the GF11 processors. Identifying which data is assigned to GF11 and how it is partitioned among the GF11 processors, should be sufficient in most situations for a Fortran-D type compiler to successfully partition the calculations among the processors. Pragmas for specifying data layout are available in Fortran-D. The interprocessor communication patterns, currently specified by the user based on the partitioning of data and calculations, would also be automatically generated by the compiler when simple data layouts are specified using the Fortran-D pragmas. Finally, currently each GF11 arithmetic/loadstore/communication operation has to be written as a separate procedure call. Certainly these semantics would change to allow the GF11 calculations to be expressed as Fortran expressions.

For most of the 1980s, it was widely believed that SIMD machines in general are ill suited for a large class of parallel applications. With our application studies on GF11 we have refuted this belief to a significant extent. We will continue to program new types of applications on GF11 to demonstrate the versatility of this architecture, and also to refine the new design to accommodate the requirements of a wider class of applications.

6. Conclusion

GF11 has been operational with 256 processors since October 1989. The full machine became operational in October 1990. A broad range of applications have already been

programmed on GF11. The use of a non-blocking circuit switched network for interprocessor communication, a large register file, a balanced bandwidth at all hierarchies of the memory, and the ability to perform many operations concurrently in each instruction are some of the GF11 design features that allow it to sustain good performance on a variety of applications. These architectural features also greatly reduce the effort required to restructure the algorithms to avoid communication/memorybandwidth bottle-necks. GF11 sustains good performance on not only the applications that are believed to be well suited for SIMD machines, but also on applications which are widely believed to be ill suited for SIMD machines.

Acknowledgements

We would like to acknowledge the effort of Mike Cassera, Molly Elliot, Dave George, Ed Nowicki, and Micky Tsao in making GF11 operational. Early work on implementing LU decomposition and FFT was done by Michael Witbrock and Banu Ozden, and is discussed in detail elsewhere. We would also like to thank Piero Sguazzaro for helping us with the Pam-Crash code.

References

[1] G.H. Barnes et al., The ILLIAC IV computer, *IEEE Trans. Comput.* C-17 (8) (Aug. 1968) 746–757.
[2] K.E. Batcher, Design of a massively parallel processor, *IEEE Trans. Comput.* C-29 (9) (Sept. 1980) 836–840.
[3] V.E. Benes, *Mathematical Theory of Connecting Networks and Telephone Traffic* (Academic Press, New York, 1935).
[4] J. Beetem, M. Denneau and D. Weingarten, The GF11 parallel computer, *Experimental Parallel Processing Architectures,* J.J. Dongarra, ed. (Elsevier Science, Amsterdam, 1987).
[5] J. Beetem, M. Denneau and D. Weingarten, The GF11 supercomputer, in: *Proc. 2th Internat. Symp. Computer Architecture,* IEEE Comp. Soc. (Jun. 1985) 108–115.
[6] W. Chiang and K. Prendergast, Numerical study of a two-fluid hydrodynamic model of the interstellar medium and population I stars, *Astrophysical J.* 297 (Oct. 1985) 507–530.
[7] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken and T. Blackadar, Performance measurements on a 128-node butterfly parallel processor, in: *Proc. 1985 Internat. Conf. on Parallel Processing,* IEEE Comp. Soc. (Aug. 22–23, 1985) 531–540.
[8] D.M. Dias and J.R. Jump, Packet switching interconnection networks for modular systems, *COMPUTER* 14 (12) (Dec. 1981) 43–54.
[9] P.M. Flanders et al., Efficient high speed computing with distributed array processor, in: *High Speed Computer and Algorithm Organization,* Kuck, Lawrie, and Sameh, eds. (Acedemic Press, New York, 1977).
[10] M.J. Flynn, Very high speed computers, *Proc. IEEE* 54 (Dec. 1966) 1901–1909.
[11] G.C. Fox, What have we learnt from using real parallel machines to solve real problems?, Caltech. Report C^2P-522.
[12] J.P. Hayes et al., A micro processor-based hypercube supercomputer, *IEEE Micro* 6 (5) (Oct. 1986) 6–17.
[13] T. Jones, Engineering design of the Convex C2, *COMPUTER* 22 (1) (Jan. 1989) 36–44.
[14] M. Kumar, Supporting broadcast connections in Benes networks, IBM Research Report RC-14063, 1988.
[15] M. Kumar and Y. Baransky, The GF11 parallel computer: Programming and performance, *Future Generation Comput. Syst.* 7 (2&3) April 1992) 169–179.
[16] R.H. Sanders and K. Prendergast, *Astrophysical J.* 188 (Mar. 1974) 489–500.
[17] R.K. Sato and P.N. Swarztrauber, Benchmarking the Connection Machine 2, *Supercomputing'88* (1988) 304–309.
[18] D.L. Slotnick, W.C. Borck and R.C. McReynolds, The SOLOMON computer, *AFIPS 1962 Fall Joint Computer Conf.* 22 (1962) 97–107.
[19] L.W. Tucker and G.G. Robertson, Architecture and application of the Connection Machine, *COMPUTER* 21 (8) (Aug. 1988) 26–38.
[20] D.L. Waltz, Applications of the Connection Machine, *COMPUTER* 20 (1) (Jan. 1987) 85–97.

[21] D. Weingarten and D. Petcher, Monte Carlo integration for lattice gauge theories with fermions, *Phys. Letters* 99B (4) (Feb. 1981) 333–338.
[22] M. Witbrock and M. Zagha, An implementation of back-propagation learning on GF11, a large SIMD parallel computer, *Parallel Comput.* 14 (3) (1990) 329–346.
[23] V. Yip and R. Elber, Calculations of a list of neighbors in molecular dynamics simulations, *J. Computat. Chem.* 10 (7) (1989) 921–927.
[24] PAM-CRASH: User Manual, Engineering Systems International, Paris (1987).