Concurrency and Computation: Practice and Experience, 16(4), April 2004, pp. 319-343.

Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines*

Xiaofang Wang and Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102
Email: ziavras@njit.edu

Abstract

Configurable computing, where hardware resources are configured appropriately to match specific hardware designs, has recently demonstrated its ability to significantly improve performance for a wide range of computation-intensive applications. With steady advances in silicon technology, as predicted by Moore's Law, Field-Programmable Gate Array (FPGA) technologies have enabled the implementation of System-On-a-Programmable-Chip (SOPC or SOC) computing platforms, which, in turn, have given a significant boost to the field of configurable computing. It is possible to implement various specialized parallel machines in a single silicon chip. In this paper, we describe our design and implementation of a parallel machine on an SOPC development board, using multiple instances of a soft IP configurable processor; we use this machine for LU factorization. LU factorization is widely used in engineering and science to solve efficiently large systems of linear equations. Our implementation facilitates the efficient solution of linear equations at a cost much lower than that of supercomputers and networks of workstations. The intricacies of our FPGA-based design are presented along with tradeoff choices made for the purpose of illustration. Performance results prove the viability of our approach.

Keywords: FPGA, LU factorization, matrix inversion, parallel processing, hardware design, SOPC/SOC.

1. Introduction

Solving a large sparse system of linear equations has always been a great challenge to conventional computing platforms, especially when operations have to be carried out in real time. An effective approach is to build high-performance parallel machines. After more than a decade of experimentation, clusters of Cray-like vector supercomputers, distributed shared-memory multicomputers employing crossbar or multistage interconnection networks, and clusters of scalar uni- and multi-processor systems dominate the high-performance computing field [1, 24]. These parallel computers have accomplished a great deal of success in solving computation-intensive problems. However, their high price, their long design and development cycles, the difficulty of sometimes programming them and the high cost of maintaining them, more often in supercomputing centers, limit their application to diverse computing fields.

* This work was supported in part by the Department of Energy under grant ER63384.

Supercomputing centers may soon become fully distributed computation brokers, serving either as "instrumentation" sites or nodes for peer-to-peer computing [1]. To make parallel computing available to the masses, all available Internet nodes in "grid computing" are candidates to solve large-scale problems in a distributed-computing fashion [22]. However, these approaches to high-performance computing are not viable for systems dedicated to a single application or for low-budget solutions.

LU factorization is a direct method that can solve large systems of linear equations that come from many important application areas, such as circuit simulation, power networks [2, 3, 28-30], structural analysis, etc. Many successful parallel LU solvers run on massively-parallel supercomputers; for example, the SuperLU algorithm has been developed for distributed-memory machines such as the Cray T3E, and for shared-memory machines such as the Cray C90 and J90, and IBM SP machines [5]. Good results for the S+ sparse LU solver have been obtained on distributed-memory machines such as the Cray T3D and T3E [6].

Real-time power flow analysis has many variations that are used frequently in the electrical power utilities industry [32]. First, for such utilities to monitor the performance of the network continuously in order to identify disturbances, such as power station failures, broken lines, and line overcharge. Second, to speed up the process of deciding to purchase electrical power from neighboring utilities according to expected customer needs and prices of available power; this process normally has a running time of several hours on PCs (personal computers). Finally, different network configurations can be tested to select the choice with the highest efficiency. Parallel processing techniques to solve power flow analysis problems have received tremendous attention in recent years [2, 3, 32-34].

On the other hand, with continuous developments in the silicon industry and advances in architecture design, FPGAs have grown to the extent that they can form SOPC computing platforms, from serving previously as simple platforms for ASIC prototyping and glue logic implementation. These advances pronounce a new promising era in the FPGA-based configurable computing field. After about a decade of active research and experimentation, configurable computing has recently proved to be a viable research avenue in accelerating algorithm execution. New generations of FPGAs have made it possible to integrate a large number of computation modules and build parallel machines in a single FPGA device. Undoubtedly, it is now the right time to reevaluate previous research efforts through the employment of this promising computing paradigm.

In this paper, we present our design and implementation of a shared-memory MIMD multiprocessor machine that uses Altera's Nios® configurable processor IP (Intellectual Property) as computing nodes [27]. The Nios embedded processor is optimized for Altera programmable logic and SOPC solutions. Altera provides a powerful, integrated-system development tool, the SOPC Builder, that supports the implementation of Nios-based embedded processor systems. We implemented our design on the Altera SOPC development board, which is populated with an EP20K1500EBC652-1x FPGA and 2

MB(ytes) of synchronous SRAM. A uniprocessor implementation with a smaller Altera device is also presented. Our highly parallel LU factorization algorithm, namely the bordered-diagonal-block sparse matrix solver for sparse matrices having unknown (i.e, not fixed) structure [2, 3], is very suitable for electrical power systems. Real electrical power systems are represented by very large sparse matrices having unknown structure, so we have adapted this algorithm for implementation on our FPGA-based parallel architecture. Our low cost, high-performance approach can improve the performance of several real-time electrical power system applications, such as the load-flow and transient stability analyses. Fields other than that of electrical power systems could also benefit from our design if the objective is to solve similar equations in reasonable running times and at dramatically reduced costs.

Our paper is organized as follows. Section 2 presents briefly the general LU factorization problem and our chosen solution. Section 3 contains an overview of our target Altera SOPC board. Section 4 presents our design of a shared-memory Nios-based multiprocessor that was implemented on this board. Section 5 illustrates further implementation issues for our design and also presents relevant execution times. Appropriate comparative analysis of the results is also included. Finally, Section 6 contains our conclusions.

2. Parallel Bordered-Diagonal-Block Sparse LU Factorization

2.1. Introduction to LU Factorization

This section presents an overview of the LU factorization problem [4, 28, 29, 30]. Solving the following system of N linear equations is the core computation of many engineering and scientific applications

$$\mathbf{A}*\mathbf{x} = \mathbf{b} \qquad (1)$$

where **A** is an N x N nonsingular matrix, **x** is a vector of N unknowns, and **b** is a given vector of length N. The solvers for this equation come mainly in two forms: direct [4] and iterative [15].

One of the classic direct methods is LU factorization, which works as follows. We first factorize **A** so that

$$\mathbf{A}=\mathbf{L}*\mathbf{U} \qquad (2)$$

where **L** is a lower triangular matrix and **U** is an upper triangular matrix. Once the elements in **L** and **U** are determined, the unknown vector **x** in (1) can be identified by forward reduction and backward substitution, respectively, using the two equations $\mathbf{L}*\mathbf{y}=\mathbf{b}$ and $\mathbf{U}*\mathbf{x}=\mathbf{y}$. LU factorization obviously has dramatically reduced costs compared to actual matrix inversion, especially for large matrices. As long as there is a solution to

the system of equations, it will be found. Moreover, the factorization result can be used repeatedly after the right hand vector has changed.

Matrix inverses are not generally used to solve systems of linear equations. LU factorization followed by forward reduction and backward substitution is a more numerically stable technique because every nonsingular matrix possesses an LU decomposition. The complexities are $\Theta(N^3)$ for LU factorization, and $\Theta(N^2)$ for forward reduction and backward substitution, so the total time needed to solve the system of linear equations with LU decomposition is $\Theta(N^3)$ [30]. Also, LU factorization saves space because the original space storing **A** is used to store **L** and **U**. In contrast, standard matrix inversion requires time $O(N^3)$ and much more space.

There are numerous variants to LU factorization. If we assume that **L** has all 1's on the diagonal and also write equation (2) in matrix form, as in

$$\begin{pmatrix} A_{11} & A_{12} & \dots & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & \dots & A_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ A_{N1} & A_{N2} & \dots & \dots & A_{NN} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & \dots & 0 \\ L_{21} & 1 & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ L_{N1} & L_{N2} & \dots & \dots & 1 \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} & \dots & \dots & U_{1N} \\ 0 & U_{22} & \dots & \dots & U_{2N} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & \dots & U_{NN} \end{pmatrix} \quad (3)$$

then we can derive the following equations for the widely used "Doolittle LU factorization algorithm" [4] that determines the matrix elements on the i^{th} row, where i assumes all values in [1,N]:

$$L_{ij} = \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} * U_{kj} \right) * \frac{1}{U_{jj}} \quad \text{for } j \in [1, i\text{-}1] \quad (4)$$

$$U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} * U_{kj} \quad \text{for } j \in [i, N] \quad (5)$$

Observing the structures of **L** and **U**, we can see that there is no need to store **L**, **U**, and **A** separately. We can use only one matrix **A** to store all three matrices. During the factorization, the modified elements in matrix **A** are destroyed and replaced with **L** and **U**. The diagonal of matrix **L** always contains all 1's and is not stored explicitly.

Another commonly used algorithm, namely "Crout factorization" [4], is similar to Doolittle factorization except that we use $U_{kk}=1$ instead of $L_{kk}=1$, for all $i \le k \le N$. From equation (3), we can then derive the following expressions for **L** and **U**, for the j^{th} step of the execution, where j assumes all values in [1,N]:

$$L_{ij} = A_{ij} - \sum_{k=1}^{j-1} L_{ik} * U_{kj} \qquad \text{for } j \in [1, i-1] \qquad (6)$$

$$U_{ij} = \left(A_{ij} - \sum_{k=1}^{i-1} L_{ik} * U_{kj} \right) * \frac{1}{L_{jj}} \qquad \text{for } j \in [i, N] \qquad (7)$$

From equations (4) to (7), we can observe that the Doolittle and Crout methods can benefit from storing the matrix in the row and column order, respectively, in relation to fast matrix accesses. Since our matrices are stored in the row order, it is more efficient to employ the Doolittle method for those parts of our LU factorization that require the application of conventional LU factorization. This is our choice in this paper.

2.2. Main Issues with Sparse LU Factorization

2.2.1. Evaluation Sequence in LU Factorization

From the **L** and **U** equations (4 and 5, respectively), we can see that before we can calculate the k^{th} row and column elements respectively, all calculations in the previous $(k-1)^{th}$ step must have been already completed; thus, all nonzero elements on the preceding rows and columns have to be available before the k^{th} loop step begins. Let us assume a 5 x 5 matrix to illustrate the precedence relation in LU factorization. Assuming the third step (k=3), we need to update all elements in the dotted rectangular shown in Figure 1. For the new value of A_{43} (actually L_{43}), we have the following expression:

$$L_{43} = (A_{43} - L_{41} * U_{13} - L_{42} * U_{23}) / U_{33}$$

$$\begin{pmatrix} U_{11} & U_{12} & U_{13} & A_{14} & A_{15} \\ L_{21} & A_{22} & U_{23} & A_{24} & A_{25} \\ L_{31} & L_{32} & U_{33} & A_{34} & A_{35} \\ L_{41} & L_{42} & A_{43} & A_{44} & A_{45} \\ A_{51} & A_{52} & A_{53} & A_{54} & A_{55} \end{pmatrix}$$

Figure 1. Precedence relations in LU factorization

Hence, we need to know all the 4^{th} row elements that are to the left of A_{43} and all 3^{rd} column elements that are above A_{33}.

If the matrix elements are distributed to different processors of a parallel computer, then frequent communication among the processors is required, which reduces the efficiency of parallel algorithms and also increases the hardware complexity of custom-made parallel machines.

2.2.2. Pivoting

We can observe from equation (4) that this algorithm is prone to numerical inaccuracies if some U_{jj}'s are very small. Of course, a major problem appears if any of the U_{jj}'s are zeros. To maintain numerical stability during factorization, pivoting is usually applied by rearranging the rows or columns of **A**. Row pivoting chooses the largest element on the k^{th} row of $A^{(k)}$ as the new diagonal element, while column pivoting chooses the largest entry on the k^{th} column of $A^{(k)}$, where $A^{(k)}$ is the A matrix in the beginning of step k. In the case of full pivoting, we choose the largest element on the k^{th} row and column. Because pivoting is dynamically determined during factorization, it greatly increases the complexity of parallel sparse LU factorization. This problem is further exacerbated if dynamic data structures are employed to store sparse matrices. For sparse matrices, the structures of **L** and **U** (that is, the location of non-zero elements) cannot be determined precisely without performing actual factorization. In SuperLU, static symbolic LU factorization is performed in order to determine in advance all possible fill-ins (positions of zeros in the original matrix that will be reproduced with non-zero elements during LU factorization), before actual LU factorization takes place [5]. Fortunately, electrical power systems employ symmetric positive definite matrices which are also diagonally dominant, so pivoting is not often required. Because we do not consider pivoting during LU factorization, we can use static data structures where all fill-ins are predetermined.

2.2.3. An Overview of the Bordered-Diagonal-Block (BDB) algorithm

Electrical power flow analysis is based on the line admittance matrix, which is a highly sparse matrix. In the admittance matrix, off-diagonal non-zero elements represent branch buses. The larger the power network is, the more sparse the matrix (i.e., the smaller the percentage of non-zero elements). For real electrical power systems, the non-zero elements in a 3000 x 3000 matrix occupy only about two percent of the matrix positions. A sparse matrix offers the advantage of reduced storage space. However, during sparse LU factorization, some of the zeros may become non-zeros, resulting in several fill-ins. So a dynamic data structure is normally required to house the fill-ins during factorization. Moreover, as discussed above, the fill-ins increase the complexity of parallel implementations.

The main aim of ordering a sparse matrix is to reduce the number of fill-ins during factorization [2-5, 30, 32, 33, 36] The ordering is to generate a permutation of the original matrix so that the permuted matrix results in a stable solution that also increases parallelism. Parallel algorithms normally include a matrix reordering phase that attempts to maximize the efficiency of the implementation. Because we do not consider pivoting during factorization, we can use static memory storage structures and the reordering can be carried out before LU factorization. Also, by ordering a sparse matrix into special forms, such as the banded, envelope, block tri-diagonal, bordered-block-triangular, and *bordered-diagonal-block (BDB)* forms, entire independent portions of a sparse matrix can be factored in parallel. Significant efforts have been made to develop efficient algorithms specifically for such forms [2, 3, 37-39]

The most successful and widely used ordering techniques [4] are: (1) Minimum degree: The rows and/or columns of matrix $\mathbf{A^{(k)}}$ at each stage k are ordered in ascending order, and the row/column with the lowest number of non-zero entries (degree) is chosen as the k^{th} row/column in order to reduce the fill-ins in the current and, hopefully, future steps. (2) Minimum fill-in: The rows and/or columns of matrix $\mathbf{A^{(k)}}$ at each stage k are ordered in an effort to produce the minimum number of fill-ins. In our implementation, we use minimum degree ordering and node tearing algorithms [3, 4, 40] in order to get a near optimal BDB matrix.

2.3. Parallel LU Factorization of a BDB Sparse Matrix

In our implementation, we use the BDB form for the matrix (see Figure 2) as our final form of ordering. It was demonstrated elsewhere that real electrical power matrices can be ordered into this form and their parallel implementation on the Connection Machine CM-5 supercomputer resulted in significant speedup for up to 16 processors [2].

In Figure 2, the $\mathbf{B_{ij}}$'s are matrix blocks; the $\mathbf{B_{ii}}$'s are referred to as the diagonal blocks and $\mathbf{B_{in}}$ and $\mathbf{B_{nj}}$ are called right border blocks and bottom border blocks, respectively, where $i,j \in [1, n]$. The blocks $\mathbf{B_{ii}}$, $\mathbf{B_{in}}$, and $\mathbf{B_{ni}}$ are said to form a *3-block group*, where $i \in [1,n-1]$. Since all other off-diagonal blocks contain all 0's, there will be no fill-ins in these blocks during factorization and the result will have the same BDB structure. From the dependence relations, we can see that only the factorization of the last block $\mathbf{B_{nn}}$ requires the data produced in the right and bottom border blocks. All other 3-block groups can be first processed in parallel, yielding very high performance. The factorization of $\mathbf{B_{nn}}$ is the last step. To factor the last block, pairs of blocks are multiplied in parallel to produce $\mathbf{B_{nj}}=\mathbf{B_{nj}}*\mathbf{B_{jn}}$, for $j \in [1, n-1]$. Then, the summation of the lower border blocks is required to factor the last block (see equations (4) and (5)). It is accumulated by the other processors and sent to the processor assigned the last diagonal block. Thus, the BDB matrix algorithm exhibits distinct advantages for parallel implementation.

$$\begin{pmatrix} B_{11} & 0 & \dots & 0 & B_{1n} \\ 0 & B_{22} & \dots & 0 & B_{2n} \\ \vdots & 0 & \dots & 0 & \vdots \\ 0 & 0 & \dots & B_{n-1\,n-1} & B_{n-1\,n} \\ B_{n1} & B_{n2} & \dots & B_{n\,n-1} & B_{nn} \end{pmatrix}$$

Figure 2. Sparse BDB matrix

BDB sparse matrix algorithms modify conventional preprocessing phases in an attempt to introduce explicit load balancing within an ordering step for uniform workload assignment to the resources of a distributed-memory multiprocessor. A new preprocessing phase was presented in [3]. Several blocks of matrices used in electrical power systems normally follow the BDB distribution for non-zero elements. The

remaining blocks in the sparse matrix need to be reordered to produce more independent diagonal blocks, which, in turn, will reduce the number of equations in the borders of the matrix. These matrix forms are normally unchanged for non-trivial amounts of time since they represent generators of electricity and existing power distribution networks. Therefore, the extra time consumed in the matrix reordering phase is easily justifiable.

The LU factorization of the BDB sparse matrix involves four steps. (1) Factorization of the independent blocks. (2) Multiplication of the right and bottom border blocks to generate the partial sums. (3) The accumulation of the partial results for the last diagonal block. (4) Factorization of the last diagonal block using the accumulated partial results from the above steps. Figure 3 illustrates these steps.

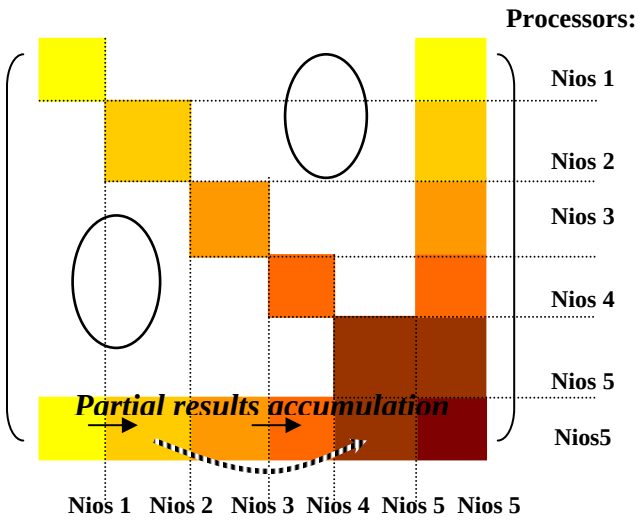

Figure 3. Parallel LU factorization of a sparse BDB matrix

To summarize, each processor contains in its local memory all data that it needs to operate on, except for the last block. Matrix data are stored in both the local memory (on-chip RAM) and the on-board SSRAM as explained in Section 4. Internal register files are used by the application. Also, no data transfers are required for the updates in the independent blocks and borders. In fact, all respective calculations can be carried out in parallel. Each of the processes for the LU factorization of the 3-block groups and the updates in the lower-right-corner block can be carried out in parallel; the former set of processes do not require any data transfers whereas the latter process necessitates the transmission of partial sums of updates to the appropriate processors that deal with calculations in the lower-right-corner block. These data transfers are rather limited and form well structured patterns in the form of binary trees (see Section 4.2.4). The most efficient algorithm should be chosen to factor the last block.

2.3.1. Preprocessing

The preprocessing phase carries out: (1) Ordering of the matrix into the BDB form. (2) Symbolic (i.e., pseudo) factorization to identify the location of fill-ins for static data structures and the actual amount of calculations corresponding to independent blocks. (3) Load balancing of the calculations among the processors. We need to emphasize here that a good load balancing technique should take into account not only the number of equations (that is, the amount of data) assigned to each processor but also the actual number of resulting operations because of non-zero elements. For good load balancing, a simulation of all the operations must be carried out in detail in the symbolic factorization phase, by taking into account all possible computations and data transfers. All three steps in the preprocessing phase can be carried out in parallel. However, the preprocessing phase is not the focus of this paper.

Let us now focus individually on each of the three preprocessing steps. Mutually independent blocks in the matrix should be identified in the ordering step. This step is based on the fact that independent sub-matrix blocks do not share edges in an undirected graph where nodes represent sub-matrices. Several techniques have been developed to implement this step. Our implementation employs the node tearing approach that has gained great popularity in circuit simulation and power analysis applications [40].

Node tearing is a very efficient partitioning technique to solve large-scale problems. If the nodes in a fine-grain undirected graph represent individual rows/columns in a symmetric matrix and the edges represent non-zero elements at the intersections of the row-column pairs represented by the incident nodes, then a grouping of rows/nodes is suggested. Edges that run between any two groups of nodes indicate coupling/inter-dependence of the corresponding groups which is expressed in the form of coupled equations. The main idea is to identify and isolate temporarily from the large problem all the coupled groups of nodes in order to generate independent sub-problems which can be solved independently. After all the sub-problems have been solved, we can solve the coupled equations. In our BDB form matrix, the independent diagonal blocks correspond to independent sub-problems, and the last (lower right) diagonal and border blocks represent the coupled nodes. Because the last block is factorized in the last step using solution data produced for preceding blocks in the matrix, we should try to make the last block as small as possible (that is, we should try to minimize the number of the coupled equations). The choice of the partitions and tear sets is based on heuristics that must take into account the physical characteristics of the matrix. In power distribution networks the buses are usually loosely interconnected, thus the node tearing algorithm can produce very good results because of the sparsity in the corresponding matrix. This matrix is symmetric in several variations of the power analysis problem. We carried out this process manually for the results presented in Section 5. We have also automated this process by running code currently residing on the host computer; this code is based on the algorithm that appeared in [40]. Within every diagonal block, we use minimum degree in the attempt to minimize the fill-ins.

To identify the location of fill-ins and also estimate the amount of required calculations in each independent block, symbolic factorization is needed. In pseudo-factorization, the entire numerical factorization process is carried out without producing any actual results.

Appropriate counters are employed to count the numbers of operations. We did not implement this step in the preprocessing phase.

From the above discussion, we can find out that the BDB matrix LU algorithm exhibits several distinct advantages for parallel implementation. First, all 3-block groups defined in Section 2.3. are mutually independent, so the LU factorization of these groups can run in parallel. In our parallel implementation, we assign to each processor distinct 3-block groups. Also, except for the right and bottom border blocks, all the off-diagonal blocks contain all 0's, so no fill-in appears outside of the diagonal blocks. Thus, we can use static data structures to represent the matrices and also distribute the independent matrix blocks among different processors.

3. Configurable Devices and Computing

3.1. SOPC/SOC devices

The terms SOPC and SOC will be used interchangeably from now on in this paper. The impact of FPGAs has been tremendous since they were first introduced by Xilinx® in 1986. In the past, FPGAs were primarily used for the rapid prototyping of digital systems and for speeding up small applications that assumed cost sensitivity and higher performance, while custom ASICs were used for high volume implementations. Due to their small chip gate count and low system speed, FPGAs were too expensive and too slow for many applications; these drawbacks were further exacerbated for entire system level design and implementation. Also, FPGA development tools were too difficult to learn and lagged in many of the features found in ASIC development systems. Newer tools have better capabilities and have attracted larger numbers of system designers.

FPGA capacities are often expressed in numbers of "system gates" that refer to the numbers of ASIC-equivalent 2-input NAND gates [26, 27]. By counting the number of system gates, we can get a sense of the amount of logic resources in an FPGA for the implementation of ASIC-equivalent designs. FPGA manufacturers normally provide the maximum number of system gates that can be used by a typical application. Current silicon manufacturing technology allows to build FPGA chips consisting of millions of system gates. This technology not only promises new levels of system integration for larger programmable chips, but also allows for more features and capabilities with reprogrammable technology. Advances in VLSI technology not only brought about multi-million gate FPGAs, but also facilitated the integration of numerous functions onto a single FPGA chip. Peripherals formerly attached to the FPGA at the board level now can be embedded into the same chip with configurable logic. According to Xilinx predictions, by 2003 the count of FPGA system gates will exceed 50 million and FPGA chips will operate at more than 500 MHz. Thus, the availability of many millions of system-level gates in FPGAs has introduced a new design paradigm, which is based on the SOC. Entire systems can be implemented on a single FPGA chip without the need for expensive non-recurring engineering charges or costly software tools. Quite often the

implementation of applications on SOCs requires the inclusion in the design of reusable Intellectual Property (IP) cores to improve productivity and reduce turnaround time; soft IP cores implement specialized units, such as FPUs (floating-point units), DSPs (digital-signal processors), and general and special-purpose processors (e.g., ARM [25], MicroBlaze [26], 80186, ARC [35]), using hardware description languages (HDLs) to uniquely define the underlying architectures.

Nevertheless, the high complexity of SOCs inadvertedly affects the complexity of pertinent application development tools. In order to deal efficiently and effectively with complex FPGA and SOC designs, and radically reduce system costs and development times, these tools should support the integration of IP cores seamlessly without reducing their performance. ASIC companies and large semiconductor vendors make available programmable-logic cores, like the VariCore EPGA IP offered by Actel. IBM has licensed FPGA technology from Xilinx for integration with its recently announced Cu-08 ASIC product offerings. The reconfigurable logic in these ASIC chips will make it possible to adopt the system's functional behavior on the fly, as needed, while still delivering high throughput because of the ASIC design. Relevant efforts initially target ease of system debugging and reduced costs in developing ASIC families. Undoutedly, these initiatives demonstrate industry convergence which is expected to make SOC approaches preeminent in the computing field. With the anticipated doubling of chip transistor densities every 18 months according to Moore's Law, our dependence on SOC designs will become even more preeminent.

3.2. Configurable Computing: An Overview

The advent of multi-million gate FPGAs has the potential to make configurable computing a flourishing field in the near future. Configurable or adaptive computing capitalizes on the static and/or run time reconfiguration of FPGA-like or switching devices and has been an active research and experimentation area ever since the introduction of commercial FPGAs [7-14, 17-20]. By loading various system configurations into FPGAs (often on the fly) as needed, the designer can achieve greater hardware functionality with the same hardware. FPGA-based (re)configurable systems can be used as specialized co-processors [16], processor-attached functional units or independent processing machines [7], attached message routers in parallel machines [17], general-purpose processors for unconventional designs [17], and general-purpose [16, 20] or specialized systems for parallel processing [12, 19]. In the past decade, FPGA-based configurable computing machines have acquired significant attention for improving the performance of algorithms in several fields, such as DSP, data communication, genetics, image processing, pattern recognition, etc. However, given the programmable nature of configurable devices, an ASIC implementation is generally faster by a factor of five to ten than its configurable counterpart [8].

Most of the configurable parallel-machine implementations currently reside on multi-FPGA systems interconnected via a specific network; ASIC components may also be present [7]. For example, Splash 2 uses 17 Xilinx XC4010s arranged in a linear array and also interconnected via a 16 x 16 crossbar [12]. For such systems, quite often the I/O

connection, and the communication between the processing elements and the host become major bottlenecks.

Research and development in configurable computing usually requires expertise in both hardware and software design. The development of automatic mapping tools is always a Herculean task for configurable systems because this is an NP-hard problem. Not only tools are needed to map and combine required hardware components onto FPGA resources, but application algorithms also have to be modified and mapped appropriately to the chosen FPGA resources in ways that yield acceptable performance [21]. Due to the difficulty of dealing with low-level hardware design, research groups have developed high-level language compilers to effectively map C/C++ code into VHDL code for targeted FPGAs [7-11, 23]. However, current compilers often require manual hardware/software partitioning and optimization, and the quality of the result in area requirements and system clock frequency is not often satisfactory [10, 11].

Dynamically reconfigurable datapaths also can be implemented with FPGAs. For example, a relatively simple co-processor for the acceleration of main computation loops in compute intensive applications was presented in [16]. This co-processor contained fixed hardware blocks and a programmable interconnect structure. A reconfigurable, dynamically programmable message router where the mapping and size of datapaths could be changing continuously was presented in [17].

Our research objective in this paper is to design a parallel machine on an SOC for the implementation of LU factorization using the BDB sparse matrix algorithm. Scalability of the algorithm-machine pair is a major objective, for the support of high-performance applications (such as power flow analysis).

3.3. The Nios Soft IP

3.3.1. An Overview

Our main implementation of LU factorization employs an Altera SOPC development board and involves many Nios processors in a shared-memory multiprocessor configuration. We have chosen a multiprocessor approach in this project in order to reduce the design and development times, and also take advantage of software available for soft processor cores (i.e., the Nios processor in this case). For special-purpose designs involving, among others, new processor development in HDL (i.e., a hardware description language) code, we either have to develop application code in assembly language targeting such a system (a quite cumbersome task indeed) or create our own compiler (which is really a Herculean task). In fact, even the task effort for the development presented in this paper has been very substantial. We spent about five months for the design of the multiprocessor architecture, the development of the application code, and the debugging of the hardware and software entities. The pioneering nature of our project necessitates that we convey such information to researchers who may attempt similar tasks in the future. Nevertheless, we are already in the process of also designing a special-purpose SIMD architecture for LU factorization

and relevant applications. One of our objectives will be to compare the performance and development costs of the design presented in this paper with the latter SIMD design when it becomes available.

An overview of Nios is pertinent. The Altera Nios RISC processor is a fully configurable soft processor running over 125 MHz in the Stratix FPGA. With the Altera-provided SOPC Builder powerful development tool, the user can build Nios-based systems on FPGAs. Combining logic, memory, and a processor core, Altera's Excalibur™ software component for embedded processor solutions allows engineers to integrate an entire system on a single programmable logic device.

The following is a quick overview of the Nios features. A block diagram of Nios is shown in Figure 4.

- General-purpose RISC microprocessor with Harvard architecture (that is, separate instruction and data buses) and a five-stage pipeline.
- 32-bit and 16-bit architectural variants of the processor.
- Complete 16-bit wide instruction set.
- Windowed register file of configurable size. 128, 256, or 512 registers may be implemented.
- The typical 32-bit Nios requires only about 2.9% of the resources contained in the EP20K1500E on the SOPC development board [27].
- Nios supports only integer arithmetic operations.

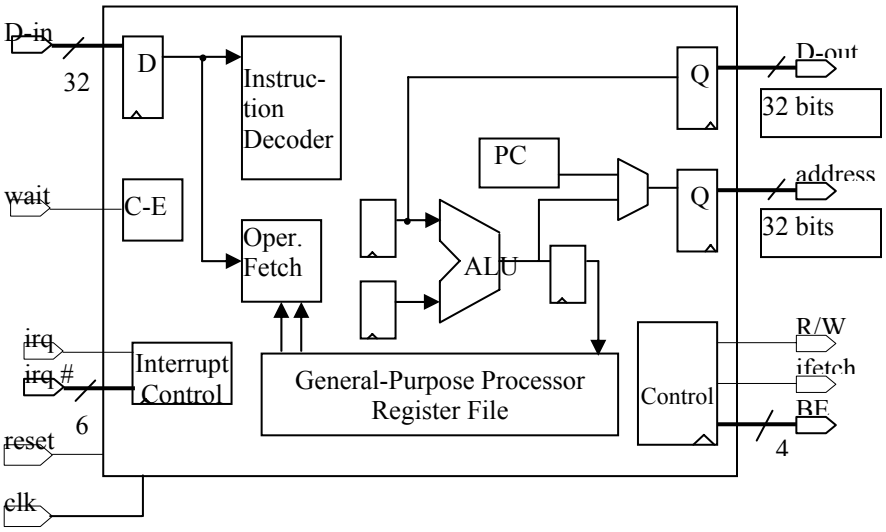

Figure 4. Altera Nios block diagram *(adapted from the Altera® web site)*. (D-in: data in, D-out: data out, R/W: read/write, irq: interrupt request, PC: program counter, C-E: clock enable, clk: clock, BE; byte enable, Oper. Fetch: operand fetch.)

From the above discussion, we can see that, many Nios processors can be implemented in a single FPGA when considering only integer operations. With the

implementation of a hardware FPU core, however, the number of implemented processors is reduced dramatically. Our design had to add a hardware FPU in each Nios in order to significantly improve the system performance. Our FPU takes almost half of the logic resources in the EP20K200E on the Nios development board.

3.3.2. Implementing Custom Instructions in Nios

A great advantage of Nios is that it allows the user to significantly increase system performance by implementing performance-critical operations through direct hardware instruction decoding. Developers are allowed to develop and include up to five parameterized custom instructions in the Nios instruction set. Additionally, user-added custom instruction logic can access memory and/or logic outside of the Nios system. By using a prefix variable, the actual number of custom instructions is only limited by the available device resources.

3.3.3. The Altera Avalon® Bus and Implementing Peripherals

SOPC boards contain several peripherals. In promoting IP-based designs, we need to enable interconnectivity with a common bus protocol. In Nios-based systems, the Nios processor(s) and other peripherals are interconnected by the multi-mastering Avalon bus. Unlike traditional shared bus protocols, the Avalon bus is a parameterized, fully connected bus that supports simultaneous transactions for all bus masters, and automatically includes arbitration for peripherals and/or memory interfaces that are shared among masters. This simultaneous multi-master architecture offers great throughput performance compared to a traditional, shared bus architecture. Each recognized SOPC component is described by a Peripheral Template File (PTF) file in the library. The users can design their component in compliance to the specifications in the PTF and integrate them into the set of Nios peripherals.

Nevertheless, a major problem arose during our implementation of the shared-memory multiprocessor. The Avalon bus tri-state bridge seems to have some problem when several Nios processors simultaneously access the data area in the off-chip SSRAM. During the LU factorization, we found out that the data read/write operations in the SSRAM memory became very unpredictable. We should mention, however, that we also tested our machine with some programs that did not have a lot of off-chip memory accesses, and the Avalon bus tri-state bridge worked well. To rectify this problem for our application that requires numerous off-chip SSRAM accesses, a sixth Nios control processor was used to prefetch data needed by the five Nios computation processors; prefetched data were stored into the latter's local memory. To speedup our application, we attempted to overlap as much as possible internal FPGA operations with external SSRAM memory accesses. More recent versions of the Avalon bus should work well.

3.3.4. The Nios Development Board

In order to facilitate Nios-based designs, Altera introduced an FPGA-based hardware development board along with the Nios development software. The Nios development

board is populated with an APEX20KE (EP20K200EFC484-2x) FPGA, which has 8,320 logic elements and 106,496 bits of on-chip RAM. It also includes 256 KB(ytes) of zero-wait-state SRAM (in two 64 K x 16-bit chips) that can be used as the program memory. The two SRAM chips are asynchronous to the Nios and all the memory operations are completed in one clock cycle. The data and address buses are shared between the two memory chips, which are different from the SOPC development board that will be introduced later. If the SRAM is not enough for the user program, the board provides a 144-pin SODIMM socket that is compatible with standard single-data-rate, 64-bit-wide SDRAM modules and can be used to expand the program memory. The 1MB on-board flash memory can be used to store the user configuration when the power is down. The configuration can be set to be loaded into the FPGA automatically using the configuration utility residing in the on-board EPROM.

The Nios board communicates with the host through an on-board RS-232 serial port. A JTAG connector is used for programming the on-board APEX device and the configuration controller.

3.3.5. Integrating an FPU with Nios

Many scientific computations, such as LU factorization for power analysis, require floating-point arithmetic to deal with large dynamic data ranges. However, FPUs have been rarely introduced in configurable machines. The most important reason is the space required for the FPU implementation due to the complexity of floating-point operations; limited numbers of resources were available in older FPGAs. The implementation of FPUs on FPGAs is feasible nowadays because of increased numbers of available resources.

We implemented a whole set of single-precision FPU instructions based on the IEEE 754 standard and an earlier FPU design. The FPU instructions are ported into the Nios system as four user instructions. The performance of our FPU is listed in Table 1. We did not purchase a commercial FPU soft core because of their very high cost. Table 2 shows a comparison of the execution times for floating-point operations implemented in software and hardware, respectively.

Table 1. Performance of the FPU for the APEX20K FPGA

	Performance in APEX20K	Resources (Logic Elements)	Clock Cycles*
Adder/Subtractor	51 MHz	696	7
Multiplier	40 MHz	2630	5
Divider	39 MHz	1028	50

Note: In order to guarantee that Nios can get the FPU result under any circumstances, we introduced an extra cycle in porting the FPU logic into Nios systems. This consideration was based on our experiments.

Table2. Execution time of software and hardware floating-point operations on the Nios system

Operations	Software Library Macros (Clock cycles)	Hardware FPU (Clock cycles)	Speedup
Addition/Subtraction	770	19	40.5
Multiplication	2976	16	186
Division	1137	51	22.3

The data in the hardware FPU column of Table 2 are the total times for a Nios processor to complete the entire instruction, including fetching and decoding times. From Table 2, we can see that the hardware implementation of floating-point arithmetic can greatly improve the performance of algorithms.

4. Design and Implementation

4.1. Sequential Implementation

We first implemented a single Nios system with an FPU on the Nios development board, for sequential LU factorization. The following is the configuration of our uniprocessor system:

- 32-bit Nios processor.
- 128 registers.
- Software multiplication is chosen (because of the hardwired implementation of the FPU, we do not need to employ the ALU embedded fixed-point multiplier).
- 1 KB of on-chip ROM to store the control program.
- 4 KB of on-chip RAM for program and data.

The total number of logic elements used is 5,900 and the system can run with a frequency of up to 40 MHz. However, our Nios board has a fixed frequency of 33.33 MHz. Table 3 shows the results for various matrix sizes and the respective speedups when compared to implementations that employ software floating-point operations.

Table 3. Comparison of uni-processor execution times (expressed in numbers of clock cycles) for various matrix sizes

Matrix Size	Software FP (clock cycles)	Hardware FP (clock cycles)	Speedup
24 x 24	3,328,615	203,567	16.35
36 x 36	11,489,708	634,137	18.12
64 x 64	70,502,845	3,326,162	21.20
96 x 96	228,013,002	10,935,456	20.85
102 x 102	274,946,133	13,076,173	21.03

With the Nios-based development board, we produced the preliminary results of LU factorization and acquired valuable design experience. We demonstrated that the

hardware FPU could significantly improve the performance of our implementation. However, due to resource restrictions, we could implement only one Nios processor with a hardware FPU. So, we then targeted our parallel design to the higher capacity Altera SOPC development board.

4.2. Multiprocessor and SOPC Implementations

Unlike the Nios development board, for which Altera provides all the documentation and fully supports it, the only documentation for the SOPC development board is the user guide. This board is populated with the biggest APEX20KE FPGA device EP20K1500EBC652-1x, which has 51,840 logic elements and 442,368 bits of on-chip memory. The board also contains two banks of SRAM memory chips with a total size of 2 MB. Each SSRAM (synchronous, static random-access memory) chip has its own data and address buses, which is a great advantage for our parallel implementation. We will discuss this later.

The memory chips on the two Altera boards are different. For the zero-wait-state SRAM on the Nios board, if the processor is rather slow (e.g., the Nios processor in our implementation that has a clock period of about 25ns) then this memory feeds the processor with data very efficiently. The zero-wait state technology supports consecutive burst read cycles by eliminating idle bus turnaround cycles. However, it requires one or two extra idle clock cycles to avoid contention when transitioning from a write to a read operation and vice versa, thus it eventually increases the duration of memory access cycles. For fast processors, it is good for applications that require frequent switches between read and write memory operations. The SOPC board has two synchronous, pipeline-burst SRAM chips (SSRAMs). Unlike the zero-wait-state SRAM on the Nios board for which all operations may take one cycle, the SSRAM chips on the SOPC board deliver data in 3-1-1-1 (read) or 1-1-1-1 (write) cycles in the burst mode, where 3-1-1-1 means that the first word takes 3 cycles and succeeding accesses consume just one cycle. There are two wait states for the first read operation. This explains why we consume more cycles when we use the on-board SSRAM memory as the program memory on the SOPC board. We compared the performance of our programs on the Nios and SOPC boards; the results are listed in Table 4. We designed the interface of the SSRAM to Nios on the SOPC board and implemented it as a standard SOPC builder library component.

Table 4. Execution times (expressed in numbers of clock cycles) on the Nios and SOPC boards for uni-processor implementations

Programs	Nios Board		SOPC Board	
	SW FP	HW FPU	SW FP	HW FPU
Multiplication of two floating-point numbers	2976	16	4376	33
LU factorization of 5 x 5	45,168	4583	78,785	7664
LU factorization of 30 x 30	7,570,660	351,843	13,592,766	674,385

From the above table, we can see that almost all the programs take 70 percent more time to run on the SOPC board than on the Nios board due to the larger SSRAM read wait states of the SOPC board.

4.2.1. Multiprocessor Architecture Design

As mentioned earlier, due to resource limitations on the Nios development board, we can only use it to run sequential LU factorization. Thus, we designed and implemented a parallel Nios-based configurable MIMD machine on the SOPC board. Figure 5 is the block diagram of our parallel machine that was configured to contain five Nios processors.

4.2.2. The Configuration of Nios Processors

Before we designed the parallel Nios system, we carefully calculated the workload of every Nios, and the requirements for program and data memories. Our LU factorization algorithm for independent blocks assumed dense submatrices, so the performance could be estimated. Because every Nios with hardware FPU requires about 6,300 logic elements and we totally have 51,840 logic elements inside the FPGA, we decided to implement five computation Nios and a separate control Nios. In order to allow for some flexibility in software mapping and routing, and to also guarantee the system clock frequency, we did not use all available logic elements in our design. The boot program was written in assembly language, had size less than 1 KB, and was stored in a 1 KB on-chip ROM. The SOPC board provides about 50 KB of on-chip memory and each Nios CPU uses about 1 KB for its register file (with the choice of 128 registers), so we assigned every Nios 6 KB of on-chip RAM. The control program stored in the on-chip ROM of each Nios processor guides every Nios. Whenever the power is turned on or the system is reset, the embedded control program prepares the processor for executing our application program. Table 5 shows the configuration of the Nios processors in our implementation.

Table 5. The configuration of the Nios processors in our multiprocessor design

CPU	Nios 1-5 (Computation)	Nios 6 (Control)
On-chip RAM (Intermediate Data)	6 KB	8 KB
On-Chip ROM (Boot Program)	1 KB	1 KB
SRAM size (Program and data memory)	192 KB	640 KB
Registers	128	256
Hardware FPU	Yes	No
Hardware Multiplier (MUL)	Yes	Yes
UART	No	Yes
Timer	No	Yes

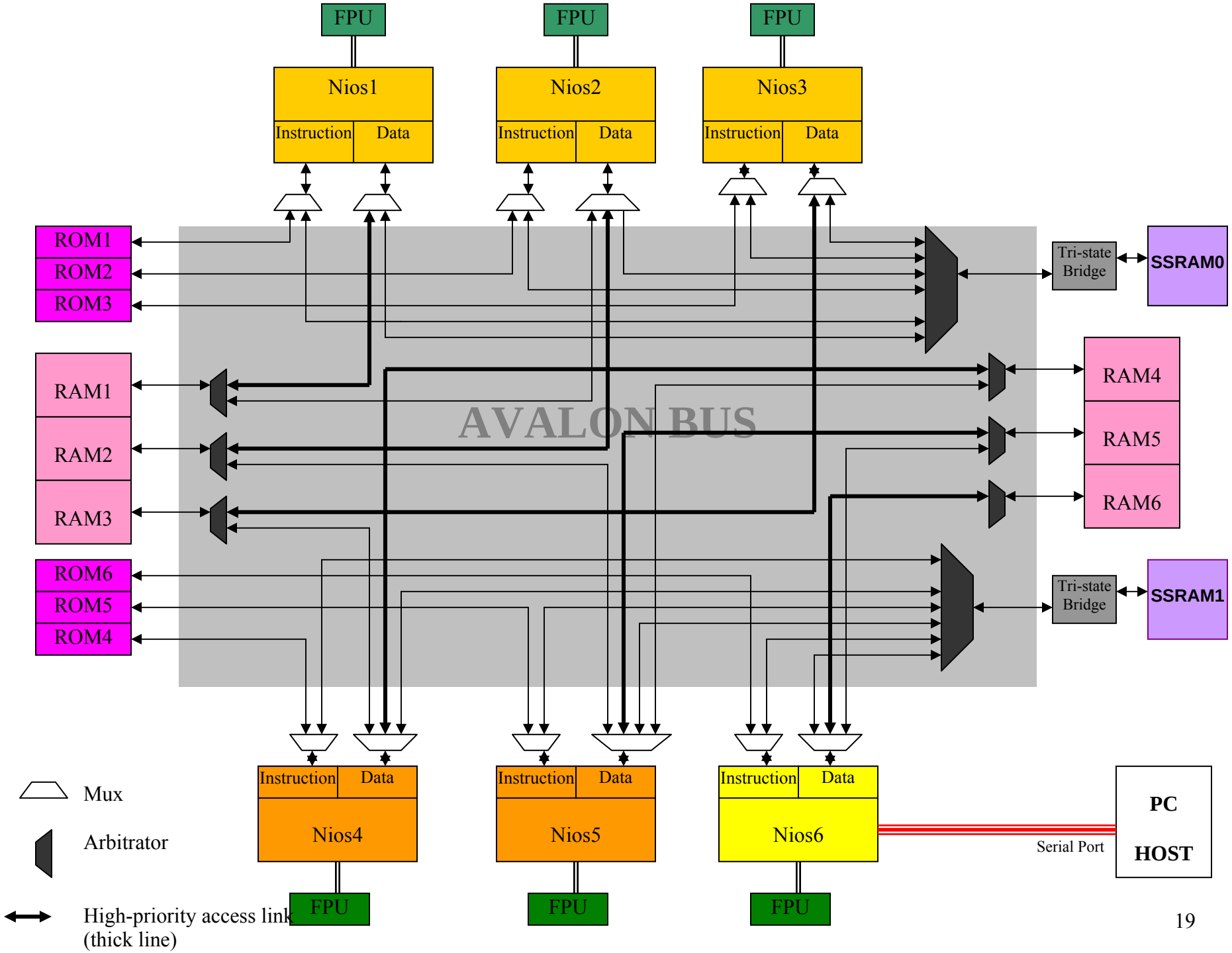

Figure 5. The architecture of our parallel machine

4.2.3. SSRAM Architecture

The two SSRAM memory chips on the SOPC board have separate address and data buses, and control signal channels. This architecture improves the system frequency and increases the memory throughput. Otherwise, with six Nios simultaneously accessing the SSRAM, the SSRAM arbitration would slowdown significantly the system's operation. We divided the SSRAM memory space into segments and assigned the same amount of memory to each Nios for main program and data needs, as shown in Figure 6. In most modern configurable machines, all SSRAM chips have their own separate data and control paths.

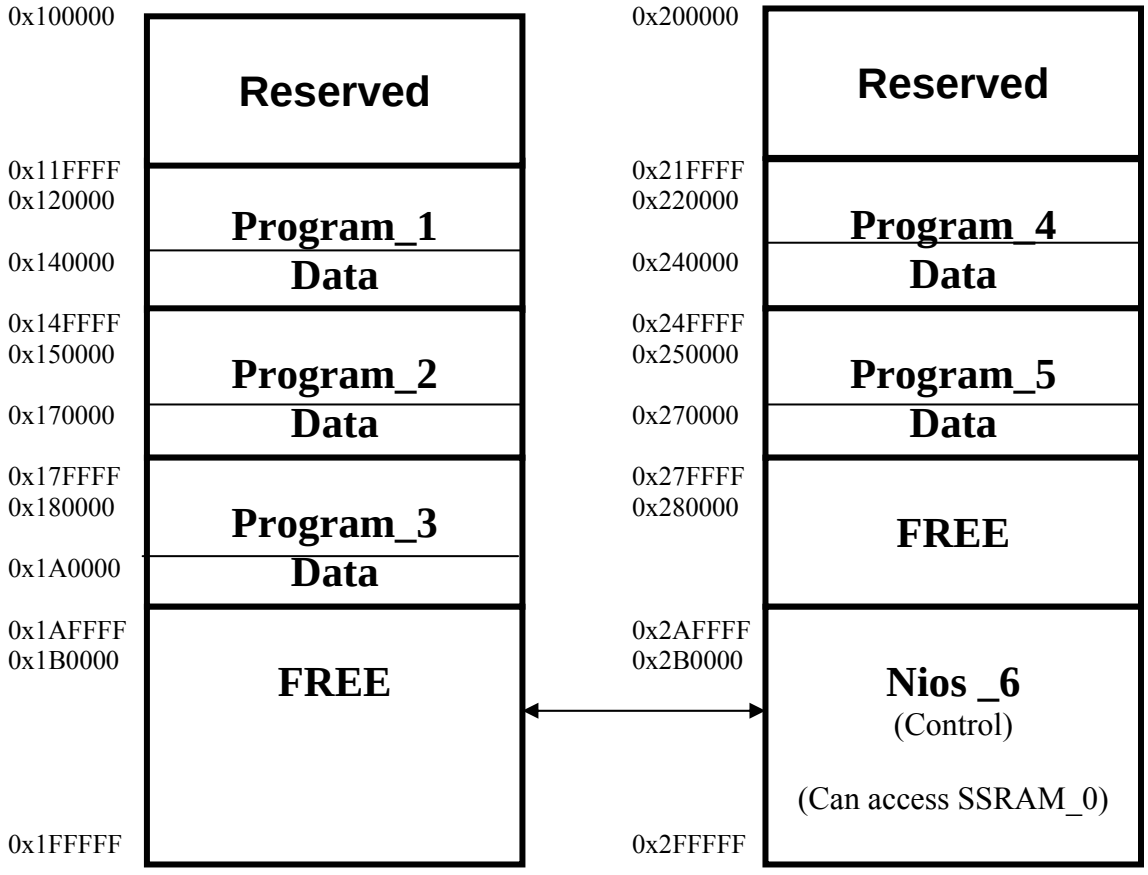

Figure 6. SSRAM memory assignments in the parallel machine

4.2.4. Partial Results

The partial results for the factorization of the last block in the lower right corner of the reordered matrix are accumulated in the following way. These three communication steps determine the required connectivity of the Nios processors.

(1) Nios 1 + Nios 2 -> Nios 2; Nios 3 +Nios 4 -> Nios 4;

(2) Nios 2 + Nios 5 -> Nios 5;

(3) Nios 4 + Nios 5 ->Nios 5.

Where -> points to the destination.

The complete schedule of operations is shown in Figure 7.

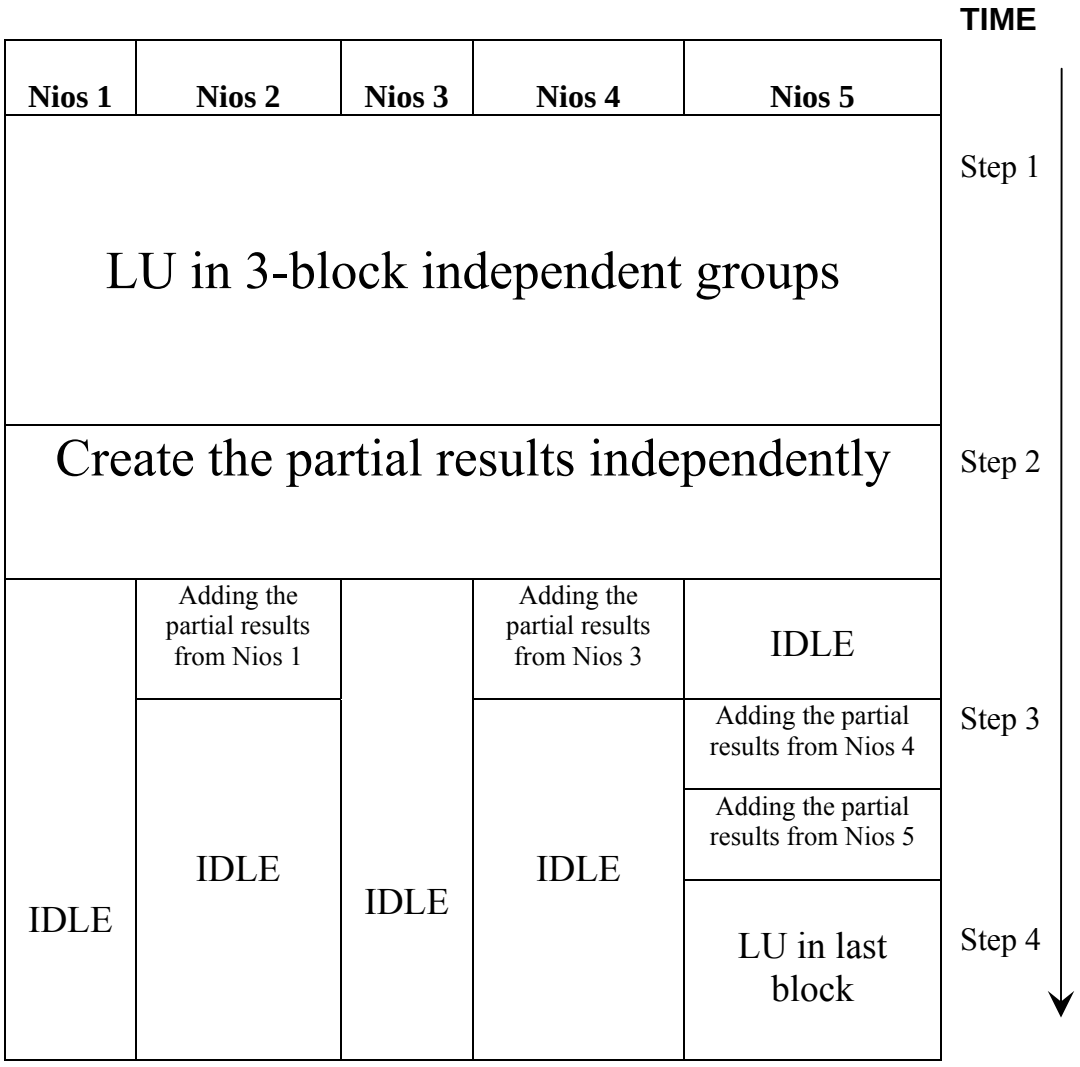

Figure 7. Scheduling the operations for parallel BDB sparse matrix LU factorization

4.2.5. More Implementation Issues

For the design of all the hardware in this project, we used the VHDL language. The boot-up program for the multiprocessor system and the control programs for all Nios processors were written in the Nios assembly language. The LU factorization programs were written in the C language. The size of the boot-up and control programs for the Nios processors 1 through 5 are about 1 Kbyte; they are about 3 Kbytes for Nios 6. The application C code is about 100 Kbytes; there is no big difference of application code sizes for different matrix sizes. We used Nios 6 for system debugging and control. This processor is connected to the on-board LEDs, LCD and UART. Through the UART this processor can interact with the host PC computer using a monitor program. We also included in our implementation Nios debug cores provided by Altera in order to be able to debug and dump the contents of Nios registers into specified memory locations. They were read back using the Nios 6 monitor program. We did not use any third party debugging tools because they are not normally designed for multiprocessor implementations.

5. Performance Results and Comparisons

The size of the independent diagonal blocks is determined during the heuristics-based reordering phase based on the number of the processors and the physical structure of the original matrix. During our experimentation with the matrices in Table 7, the number of the independent diagonal blocks is the same as the number of computation processors, namely 5. With increases in the matrix size and reduction of its sparsity, we may make the number of the independent diagonal blocks a multiple of the number of computation processors in order to assign every processor several 3-block groups. The last processor (Nios 5) is assigned one 3-block group as well as the last diagonal (lower-right-corner) block on which it performs sequential LU factorization.

Table 6 shows the expected performance of our implementation and the actual execution times on our parallel machine for a 30 x 30 matrix. Detailed execution times are included to show that different steps with the same asymptotic complexity have quite different execution times; the details in these steps must be taken into account by load balancing techniques for problems assuming large matrices.

Table 6. The computation complexity of the parallel algorithm and detailed results

Step	1	2	3	4
Complexity	$O((\frac{N}{p})^3)$	$O((\frac{N}{p})^3)$	$O((\frac{N}{p})^2 * \log_2 p)$	$O((\frac{N}{p})^3)$
Clock Cycles for 30x30	21,599	8914	8885	9312

A brief explanation of the complexities shown in Table 6 follows. We assume that:

- the size of the matrix is N x N;
- p represents the number of processors that participate in the computation;
- the size of the largest diagonal block is k x k;
- the size of the last diagonal block in the reordered matrix is O(m) x O(m).

Without loss of generality, we can assume that $O(k)=O(\frac{N}{p})$ and O(m)=O(k) for simple attempts to equibalance the work load .

Step 1: For the LU factorization of the 3–block independent groups in parallel, the maximum execution time depends on the largest diagonal block of size k x k. So the computation complexity is $O((k+m)^3-m^3) = O(k^3) = O((\frac{N}{p})^3)$.

Step 2: The multiplication of the right and bottom border blocks in parallel has a complexity of $O(k^3)= O((\frac{N}{p})^3)$.

Step 3: A binary tree of processors is formed to carry out the additions of O(k) x O(k) matrices in this phase of the algorithm. For a total number of p processors, this phase consumes time $O(k^2\log_2 p) = O((\frac{N}{p})^2 \log_2 p)$.

Step 4: The LU factorization of the last diagonal block in the lower right corner has a complexity of $O(k^3) = O((\frac{N}{p})^3)$.

Therefore, the total computation time is $O(k^3+k^3+k^2\log_2 p+k^3)=O((\frac{N}{p})^3+(\frac{N}{p})^2\log_2 p)$ $=O((\frac{N}{p})^3)$, assuming that $O(N) > O(p \log_2 p)$ for large matrices and medium granularity parallel systems.

Table 7 shows more performance results involving various matrix sizes.

During our experimentation we observed that, when the size of blocks assigned to processors is a power of two, then each Nios works more efficiently. Higher efficiency in these cases may be the result of smoother pipelined access of matrix data in the memory. These high speedups prove the viability of our approach in solving the LU factorization problem for sparse matrices.

Table 7. Execution times of our parallel LU factorization algorithm with a clock frequency of 40 MHz

Matrix Size / Total Cycles	24 x 24	30 x 30	36 x 36	42 x 42	48 x 48	54 x 54	96 x 96	102 x 102
Multi-processor	22,041	48,710	38,274	55,618	106,909	177,510	624,415	852,002
Uni-processor	79,630	165,141	136,037	202,878	414,874	671,711	2,511,122	3,404,160
Speedup	3.61	3.39	3.55	3.65	3.88	3.78	4.02	3.995

We also applied our approach to a power flow analysis problem that uses real data for the IEEE 118-bus test system. The data representation involves a 118 x 118 admittance sparse matrix. We reordered the B matrix used in the decoupled load flow iterative algorithm in the PC host to make it appropriate for our LU factorization algorithm. Execution times for the SOPC board are shown in Table 8. The time corresponding to the reordering is not included in the results. Our implementation shows the viability of our approach that employs FPGAs to efficiently solve this problem at low cost.

Nevertheless, a better FPGA-based development board could give us better results. We found out that the major bottleneck in the system lies in the interface to the two SSRAM chips. Since the application programs do not fit in the on-chip memory, all processors use the on-board SSRAMs as the instruction and data memory; for the sake of scalability and flexibility, we should not actually rely on the on-chip memory to hold the application programs. If we could design our own interface to the SSRAM chips, we should let the number of SSRAM chips be equal to the number of processors in our implementation; alternatively, we should include as many SSARM chips as possible for a given number of FPGA pins. This approach can reduce the complexity of the arbitrator for accessing the SSRAM chips and this, in turn, can result in higher system frequency and improved throughout. The FPGA on our SOPC board has a maximum of 488 I/O pins that can be used in user designs; this number is adequate for our current design.

Our SOPC system has very low cost compared to large parallel machines and supercomputers. The cost of our system was less than three thousand dollars more than a year ago and it is, therefore, many orders of magnitude lower than the cost of the latter systems. In fact, such a direct comparison is not fair at this time because of many reasons. First, FPGAs have not been perfected yet because they represent relatively new technology. Second, FPGAs do not currently contain enough resources to help us implement large parallel designs; however, this is expected to change in the future primarily because of Moore's Law. Third, the design tools for FPGAs are not really very good at this time. Nevertheless, more advanced tools will become available in the future

as more designers attempt to use FPGAs for complex designs. Driven by expected advances predicted by Moore's Law, many researchers have recently focused on the design of multiprocessor chips [31]. Such chips will often have to be prototyped on FPGAs. Fourth, our design does not give very good performance because it does not include a good FPU. The design of a very good superpipelined FPU is not a trivial task, free HDL for very good FPUs is not available to the public, and the price of such a soft IP is prohibitively high (more than $10,000). Our goal in the university environment is to prove the viability of our design concepts and approaches; we do not have to acquire the best possible FPU (in the form of a soft IP) in order to compete directly with commercial supercomputers. Fifth, better performance can be achieved by implementing multi-FPGA boards targeting specific applications. Finally, specialized FPGA designs do not normally have compiler and other advanced software support for ease of program coding and most efficient implementation using general-purpose load balancing tools. The designer of the hardware system is normally the person who also writes the application and tries to fine-tune it for higher performance. To conclude, it is only natural that FPGAs will receive more attention by researchers in the near future as their need will become more prevalent. As a result, they will be able to improve significantly.

The cost of FPGAs is not much higher compared to conventional processor chips. A single FPGA chip has market cost comparable to a newly designed advanced processor chip. However, the cost of FPGA-based development boards is often higher than that of PCs but it is only because the use of such boards is not widespread at this time. Their slightly higher cost is the result of market forces, not actual production cost. It may even be true that the production cost of a top-of-the-line processor is much higher than the production cost of an FPGA (when considering same quantities of chips) because the FPGA has a basic structure that is replicated throughout the chip; in contrast, the processor has a non-uniform design. The R&D cost of a completely new processor is in the billions of dollars range that companies producing FPGAs cannot afford. Also, HDL implementations are highly portable to different FPGA platforms, whereas new hardware designs for PCs and comparable systems require new production lines.

6. Conclusions

In this paper, we described the design and implementation of a multiprocessor shared-

Table 8. Performance results for the IEEE 118-bus system (118 x 118 matrix)

% of non-zero elements	Block size		Execution times (ms)				
	Largest independent diagonal block	Last diagonal block	Step 1	Step 2	Step 3	Step 4	Total
3.42	23	25	18.47	12.12	2.37	8.20	41.16

memory architecture on an FPGA-based system. It can result in reasonable performance at low cost for the parallel LU factorization of sparse BDB matrices. Our results show that the new generation of SOPCs provides viable computing platforms that offer the possibility of building high-performance parallel machines in one programmable device. Because our implementation was based on a relatively slow FPGA device (the Altera EP20KE series FPGA) and a non-advanced FPU, the system frequency was not satisfactory. The new Virtex II device from Xilinx can achieve up to 420 MHz system frequency. With the doubling of the transistor density in silicon chips every 18 months, as predicted by Moore's Law, we strongly believe that this research avenue will become even more promising in the near future. Also, our work has shown that it is necessary to utilize existing IP components for the fast design of robust systems in large capacity programmable devices. The intricacies of our design were presented to guide future attempts in this research arena.

7. Acknowledgement

The authors would like to gratefully acknowledge the support provided by Altera® Corp. in the form of Nios development boards and some software. Also, the paper has been improved substantially because of the suggestions made by the four referees.

References

1. G. Bell and J. Gray, "High Performance Computing: Crays, Clusters, and Centers. What Next?" *Tech. Rep., MSR-TR-2001-76,* Microsoft Research, San Francisco, California, Aug. 2001.
2. D.P. Koester, S. Ranka, and G.C. Fox, "Parallel LU Factorization of Block-Diagonal-Bordered Sparse Matrices," *Tech. Rep. SCCS-550*, *Northeast Parallel Architectures Center*, Syracuse University, Syracuse, New York, Oct. 1994
3. D. Koester, S. Ranka, and G.C. Fox, "A Parallel Gauss-Seidel Algorithm for Sparse Power Systems Matrices," *Proc. Supercomputing '94,* Washington, D.C., Nov. 1994, pp. 184-193.
4. I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford Univ. Press, Oxford, England, 1990.
5. J. W. Demmel, J. R. Gilbert, and X. S. Li, "An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination," *SIAM J. Matrix Anal. Appl.,* Vol. 20, No. 4, 1999, pp. 915-952.
6. C. Fu, X. Jiao, and T. Yang, "Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures," *IEEE Trans. Paral. Distr. Systems,* Vol. 9, No. 2, Feb.1998, pp. 109-125.
7. K. Compton, S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Comput. Surveys*, Vol. 34, No. 2., June 2002, pp. 171 - 210.
8. R. Tessier and W. Burleson, "Reconfigurable Computing and Digital Signal Processing: A Survey," *J. VLSI Signal Proc.*, May/June 2001, pp. 7-27.

9. R. Hartenstein, “A Decade of Reconfigurable Computing: A Visionary Retrospective,” *IEEE Proc. Int. Conf. Exhib. Design Autom. Testing Europe*, Munich, Germany, 2001, pp. 135-143.
10. J. Frigo, M. Gokhale, and D. Lavenier, “Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective,” *Proc. 9th ACM/SIGDA Intern. Symp. Field Program. Gate Arrays*, Monterey, California, Febr. 2001, pp. 134-140.
11. M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski, "Stream-Oriented FPGA Computing in the Streams-C High Level Language," *Proc. 8th Annual IEEE Symp. Field-Program. Custom Comput. Machines*, Napa, California, Apr. 2000, pp. 63-69.
12. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder, *Splash 2: FPGAs In a Custom Computing Machine*, IEEE Computer Society Press, Los Alamitos, California, 1996.
13. N. Shirazi, A. Walters, and P. Athanas, **“**Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines,” *IEEE Proc. Symp. FPGAs Custom Comput. Machines,* Napa Valley, California, April 1995, pp. 155-162.
14. J. Cloutier, E. Cosatto, S. Pigeon, F. R. Boyers, and P. Y. Simard, “VIP: An FPGA-Based Processor for Image Processing and Neural networks,” *5th Intern. Conf. Microelectr. Neural Net. Fuzzy Syst.,* Lausane, Switzerland, Feb. 1996, pp. 330-336.
15. E.V. Krishnamurthy and S. G. Ziavras, “Complexity of Matrix Partitioning Schemes for g-Inversion on the Connection Machine,” *Cent. Autom. Res. and Comp. Sci. Dept. Tech. Rep.,* University of Maryland, CAR-TR-400 and CS-TR-2126, Oct. 1988.
16. Z. Huang and S. Malik, “Exploiting Operation Level Parallelism Through Dynamically Reconfigurable Datapaths,” *Design Auto. Conf.,* New Orleans, Louisiana, June 10-14, 2002, pp. 337-342.
17. S. Ingersoll and S.G. Ziavras, “Dataflow Computation with Intelligent Memories Emulated on Field-Programmable Gate Arrays (FPGAs)," *Microproc. Microsys.*, Vol. 26, No. 6, Aug. 2002, pp. 263-280.
18. T. Golota and S.G. Ziavras, “A Universal, Dynamically Adaptable and Programmable Network Router for Parallel Computers," *VLSI Design,* Vol. 12, No. 1, 2001, pp. 25-52.
19. S.G. Ziavras, “Investigation of Various Mesh Architectures with Broadcast Buses for High Performance Computing,” *VLSI Design,* Special Issue High Perform. Bus-Based Arch., *R. Lin and S. Olariu (Eds.),* Vol. 9, No. 1, Jan. 1999, pp. 29-54.
20. S.G. Ziavras, ``Scalable Multifolded Hypercubes for Versatile Parallel Computers," *Parall. Proces. Lett.*, Vol. 5, No. 2, June 1995, pp. 241-250**.**
21. S.G. Ziavras, “Efficient Mapping Algorithms for a Class of Hierarchical Systems," *IEEE Trans. Paral. Distr. Systems,* Vol. 4, No. 11, Nov. 1993, pp. 1230-1245.
22. I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, “The Physiology of the Grid: An Open Services Architecture for Distributed Systems Integration,” Draft of work in progress, http://www.globus.org/research/papers/ogsa.pdf.

23. T. Grotker, G. Martin, S. Liao, and S.Swan, *System Design with System-C, Kluwer Acad. Publ.,* Boston, Massachusetts, 2002.
24. X. Li, S. G. Ziavras, and C. N. Manikopoulos, "Parallel DSP Algorithms on TurboNet: An Experimental Hybrid Message-Passing/Shared-Memory Architecture," *Concurrency: Pract. Exper.*, Vol. 8, No. 5, June 1996, pp. 387-411.
25. *ARM Soft IP Processor*, http://www.arm.com.
26. *Xilinx Inc. Web Site*, http://www.xilinx.com.
27. *Altera Corp. Web Site*, http://www.altera.com.
28. G.T. Heydt, *Computer Analysis Methods for Power Systems*, Macmillan Publ., Basingstoke Hampshire, England, 1986.
29. J.J. Grainger and W.D. Stevenson, Jr., *Power System Analysis*, McGraw Hill Publ., 1994.
30. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw Hill Publ., New York, New York, 1994.
31. K.A. Shaw and W.J. Dally, "Migration in Single Chip Multiprocessors," *Comput. Archit. Lett.*, Vol. 1, No. 3, Nov. 2002, pp. 2-5.
32. D.J. Tylavsky, et al. "Parallel Processing in Power Systems Computation," *IEEE Trans. Power Systems*, Vol. 7, No 2, May 1992, pp. 629-638.
33. S.-L. Lin and J.E. Van Ness. " Parallel Solution of Sparse Algebraic Equations," *IEEE Trans. Power Systems*, Vol. 9, No 2, May 1994.
34. T. Feng and A.J. Flueck, "A Message-Passing Distributed-Memory Parallel Power Flow Algorithm," *IEEE Power Engin. Soc. Winter Meet.*, Vol. 1, 2002, pp. 211 –216.
35. *ARC Soft IP Processor*, http://www.arc.com.
36. A. Gupta, "WSMP-Watson Sparse Matrix Package, Part I-Direct Solution of Symmetric Sparse Systems," *IBM Research Report* RC 21886 (98462),Yorktown Heights, New York, Nov. 16, 2000.
37. A. Gomez and L.G. Franquello, "An Efficient Ordering Algorithm to Improve Sparse Vector Methods," *IEEE Trans. Power Systems*, Vol. 3, No 4, Nov. 1988, pp. 1538-1544.
38. A. Gomez and L.G. Franquello, "Node Ordering Algorithms for Sparse Vector Method Improvement," *IEEE Trans. Power Systems*, Vol. 3, No 1, Febr. 1988, pp. 73-79.
39. Y.Q. Wang and H.B. Gooi, "New Ordering Methods for Sparse Matrix Inversion via Diagonalization," *IEEE Trans. Power Systems*, Vol. 12, No 3, Aug. 1997, pp. 1298-1305.
40. L. O. Chua and L. K. Chen, "Diakoptic and Generalized Hybrid Analysis", *IEEE Trans. Circuits Systems*, Vol. 23, No. 12, 1976, pp. 694-705.