A Configurable Multiprocessor and Dynamic Load Balancing for Parallel LU Factorization*

Xiaofang Wang and Sotirios G. Ziavras
Department of Electrical and Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102
{xw23, ziavras}@njit.edu

Abstract

The exponentially increasing complexity of many scientific applications and the high cost of supercomputing force us to explore new, sustainable, and affordable high-performance computing platforms. Recent significant advances in FPGA technology and the inherent advantages of configurable logic have brought about new research efforts in the configurable computing field: parallel processing on configurable chips. We explore here parallel LU factorization of large sparse block-diagonal-bordered (BDB) matrices on a configurable multiprocessor that we have designed and implemented. A dynamic load balancing strategy is proposed and analyzed. Performance results for IEEE power test systems are provided. Our research provides evidence that configurable logic can be a viable alternative to high-performance scientific computing.

Keywords: *parallel processing, LU factorization, FPGA, multiprocessor, dynamic load balancing.*

1. Introduction

LU factorization is a fundamental method that is widely employed to solve large systems of linear equations appearing in many important application areas, such as circuit simulation, power networks, structural analysis, etc. Due to its high computational complexity, $O(N^3)$ for a dense matrix, where N x N is the size of the matrix, research efforts on parallel implementations have been ongoing for several decades. However, due to the scarce availability and the prohibitively high cost of proprietary supercomputers and other parallel machines, most of these research results are limited to a narrow range of applications. The recent shrinking of the supercomputer market makes this issue more prominent.

The desire for affordable high-performance parallel computing inspired many research efforts in recent years in COTS-based (commercial-off-the-shelf) platforms, such as symmetric multiprocessing (SMP) clusters or multiprocessors and Beowulf clusters. The community of cluster computing continues to expand, with tera-scale clusters now in production and peta-scale clusters in design. However, these approaches do not deliver the highest level of performance due to many inherent disadvantages of the underlying sequential platforms, such as the much greater communication latency between processing nodes. Moreover, the computation demands of many application algorithms increase at a higher rate than Moore's Law [3]. The different requirements of general computing and scientific applications convince us that we should not rely heavily on COTS components at the lowest architectural level for a bright future of high-performance computing.

At the same time, FPGA-based configurable computing is becoming more and more appealing and has resulted in impressive achievements for many computation-intensive applications [1-10, 14-16]. Recently available multi-million platform FPGAs with richer embedded feature sets, such as plenty of on-chip memory, DSP blocks and embedded hardware microprocessor IP cores, have made it feasible to build parallel systems on a programmable chip (PSOPC). Our specific research motivation is to build cost-effective high-performance parallel systems within FPGAs in order to enable the solution of large sparse linear systems of equations. We have implemented a scalable shared-memory multiprocessor within a single FPGA based on configurable processor IP cores; we investigated parallel LU factorization of large sparse BDB matrices on our machine [14]. The obtained performance provides evidence that FPGA-based custom configurable machines can be cost-effective platforms for high-performance scientific computing.

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

The BDB form has long been considered to be a desirable structure for large sparse matrices due to its inherent features for parallel implementation. In this form, all non-zero elements are grouped into blocks along the diagonal, and the right and bottom borders. As a result, each diagonal block and the two border blocks on the corresponding row and column can be factored independently and in parallel with all other such 3-block groups; this does not apply to the last diagonal block in the lower right corner [14]. Parallel BDB LU factorization includes a preprocessing phase where the input sparse matrix is reordered and partitioned into the BDB form. For most large sparse matrices, the obtained blocks are irregular in size and sparsity. So, the floating-point operations involved in every subtask vary and the execution times are unpredictable. Moreover, fill-ins (appearing when zero elements become non-zero elements) that occur dynamically at runtime also add more indeterminacy. Therefore, dynamic load balancing is needed.

In this paper, we further improve our hardware design for the configurable multiprocessor and explore dynamic load balancing for parallel LU factorization of large sparse BDB matrices. In contrast to our approach in [14], the factorization of the last block also is parallelized efficiently on our new architecture. We introduce the improved multiprocessor architecture and hardware design details with a focus on the memory system in Section 2. Section 3 presents parallel LU factorization of sparse BDB matrices on our machine. We explore the load balancing strategy for this algorithm on our machine in Section 4. A theoretical performance analysis is also included in this section. Experimental results for IEEE power flow test cases with up to 7 processors are presented in Section 5. A summary is given in Section 6.

2.FPGA-based configurable multiprocessors

2.1 Recent FPGA developments

With increases in logic resources per chip and improved architectures, FPGA-based configurable computing recently became very appealing and has resulted in many impressive achievements. It is anticipated that chips with 50 million gates of reconfigurable logic will be available by 2005 at substantially lower costs. Limited resources are no longer a major hurdle for the design of large FPGA-based systems. The flexibility, re-programmability and run-time reconfigurability of FPGAs have great potential to offer an alternative computing platform for high-performance computing.

Configurable RISC processor IP cores have recently become available to greatly empower FPGA-based system implementations. Configurable processors add another dimension in programmability and flexibility. We can tailor the processor to the specific requirements of the application and include only those features that are needed by the application. The instruction set architecture (ISA), register file, software development APIs (application programming interfaces), memory hierarchy and size, and communication channels can all be configured and extended. Also, standard and user customized logic engines can be easily added, modified or extended, as needed. We can identify critical instructions in the application code that affect performance the most and implement them in hardware. Configurable processor cores also provide us with more flexibility to integrate them in an SOPC environment with other IPs, compared to fixed processor cores. New generation FPGAs can host dozens of RISC processor cores, which shows the feasibility of building parallel systems in a single FPGA.

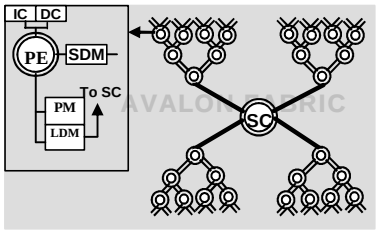
Moreover, the performance and efficiency of algorithms highly depend on their good match with the target architecture. New FPGA-based configurable computing strategies provide the system designer with several dimensions to optimize the design for application-specific performance. Full control is viable over most of the resources and enormous opportunities appear during the overall design process. However, most of the current FPGA development kits are based on similar design flows and languages with ASIC design methods. Although some groups work on high-level C/C++-like languages to close the gap between hardware and software design methodologies, the performance is often still one or two orders of magnitude lower than that of manually optimized implementations [1, 6]. There are no effective design methodologies and development tools available for this new codesign model, in particular for parallel systems. It requires the designer to be proficient in algorithms, system-level design, software/hardware partitioning, architecture design, and software/hardware coding.

2.2 Our multiprocessor architecture model

Fig. 1 shows our processor-based system model for the parallel BDB LU factorization algorithm [14]. For the sake of brevity, this algorithm is presented along with its implementation on our system in the next section. As the feature size of silicon processes enters the submicron range, the wire delay becomes even more significant compared to the logic delay, and it can even dominate the system's performance due to the reverse scaling of wires compared to transistors. The routing of chip-level and clock signals tends to become more cumbersome in complex multi-million gate SOPC designs. In contrast, our binary tree network for data communications eliminates global transfers. Our clustered binary tree topology has been chosen by the communication patterns

in our algorithm (details are in Section 3). The control channel is a star connection between the SC (system controller) and every PE. There is also a direct communication channel between the SC and every parent of the binary tree. We implemented serial and TCP connections between the multiprocessor and the host computer.

2.2.1. Processing element (PE). The PE lies in the core of any computing machine. We used a 32-bit Nios IP processor from Altera as the PE. The system controller is also implemented with Nios. The Nios RISC processor is fully configurable and runs at over 125 MHz in the Altera Stratix FPGA. It utilizes a 5- stage pipeline and conforms to a modified Harvard memory architecture. With configurable processors we need to carry out trade-offs between the processing power and resources used. A typical Nios processor in our machine consumes about 1600 logic elements in the APEX20KE device. The number of PEs in our computing model is scalable, as shown in Fig. 1. The operation of every PE is guided by the SC that utilizes the boot code in the on-chip individual program memory of PEs and its interrupt connection to individual PEs. Our multiprocessor targets in general complex matrix algorithms that require floating-point arithmetic to deal with dynamic data of wide range. Our applications also apply some trigonometric functions, such as sine and cosine, which take considerable time if implemented in software. We implemented floating-point arithmetic and these functions in hardware and interfaced the application code as custom instructions. Such hardware customization also releases many resources to the processor for other tasks. The reduced clock cycles resulted in significant improvement in the matrix operations reported in [14-16].



SC: System Controller
IC: Instruction Cache
LDM: Local Data Memory
PM: Program Memory
PE: Processing Element
DC: Data Cache
SDM: Shared Data Memory

Figure 1. Our multiprocessor architecture model

2.2.2. Memory hierarchy design. Due to the current lack of latency reducing software support for configurable machines, the memory design becomes a dominant factor in system performance. Moreover, while new silicon technology and computer architecture research facilitate faster processors, the performance gap between processors and memories tends to become larger. In our shared memory multiprocessor, the overall speedups may be quickly diminished due to severe memory contention and large system synchronization, if we rely solely on the on-board SRAM memory as the main runtime memory. Fortunately, new generation FPGAs make available large on-chip memory with wide communication channels. Our FPGA-based multiprocessor architecture capitalizes on this advantage and forms several kinds of memories in order to maximize performance. For example, we implemented a controller to oversee the system's operation, and also pre-fetch instructions and data from the on-board memory into the PEs; the latter use the on-chip memory to run the application code because of its much lower latency compared to the on-board SRAM memories.

Every PE has a local on-chip program memory and a shared-data memory. The data memory is shared with its sibling and parent, as shown in Fig. 2. The sizes of the program and data memories for each PE are determined by the available memory capacity of FPGAs and the total number of PEs. The shared-data memory improves the performance by eliminating the transfer of large blocks of data between memories. All the required interconnection between on-chip memories and/or processors is implemented based on the multi-mastering, fully connected AVALON bus of Altera. Thus, the communication bandwidth is quite large and the on-chip memory access time is only one clock cycle.

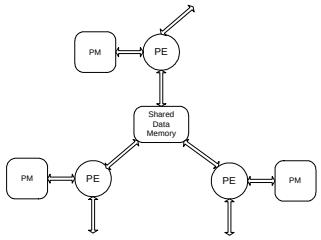


Figure 2. Memory configuration (PM: program memory)

Because it takes on the average at least 4 clock cycles to access the on-board synchronous SRAM memory, on-chip data and instruction caches are employed to reduce the memory access latency. For a fixed system, we focus on the efficient utilization of a cache of fixed size and configuration. For a configurable processor, we have choices in both hardware configuration and software optimization. In order to find an optimal cache size for our application, we compared the performance of a single processor with different instruction and cache sizes for the

LU factorization algorithm applied to a 30 x 30 matrix. We can tell from the results shown in Fig. 3 that the cache configuration can make a difference of more than 20% in performance. We used a direct-mapping cache with the write-through policy in our experiments. The number preceding "(I)" or "(D)" represents the size of the instruction or data cache, respectively, expressed in Kbytes.

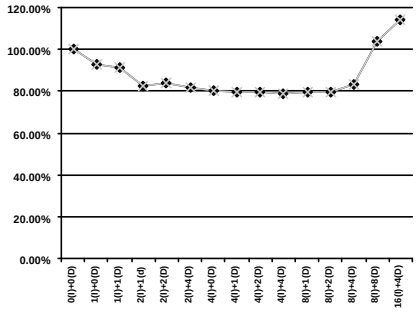


Figure 3. Relative execution time for 30 x 30 LU factorization compared to a non-cached system

3. Parallel BDB LU factorization

Many scientific and engineering problems, such as circuit simulation, applications in electric power networks and structural analysis, require to solve a large sparse system of simultaneous linear equations of the form **Ax** = **b**; **A** is an *N* x *N* nonsingular matrix, **x** is a vector of *N* unknowns and **b** is a given vector of length *N*. One of the classic and widely employed direct methods is LU factorization, which works as follows. We first factorize **A** so that **A** = **LU**, where **L** is a lower triangular matrix and **U** is an upper triangular matrix. Their elements can be determined by $L_{ij} = (A_{ij} - \sum_{k=1}^{i-1} L_{ik} * U_{kj}) * \frac{1}{U_{jj}}$, *for* $j \in [1, i-1]$ and $U_{ij} = A_{ij} - \sum_{k=1}^{i-1} L_{ik} * U_{kj}$, *for* $j \in [i, N]$**,** respectively [13]. Once **L** and **U** are formed, the unknown vector **x** can be identified by forward reduction and backward substitution, respectively, using the two equations **Ly** = **b** and **Ux** = **y**. Since LU factorization is a computation-intensive procedure, its parallel solution has been a quite active research area. Thus, plenty of parallel techniques have appeared in the literature.

Although the LU factorization of sparse matrices appears to be easier to parallelize, it suffers from a significantly unique dynamic problem corresponding to fill-ins. Our earlier research has shown that it is often inefficient to extract instruction level parallelism (ILP) in the LU factorization of sparse matrices due to irregular data dependences and the limited scalability of the parallel implementations. We believe that data partitioning is an efficient and scalable approach to parallelize LU factorization algorithms.

One of the partitioning schemes is to reorder and partition the **A** matrix into the BDB form by the node-tearing technique [12] or similar heuristics. In the BDB form shown in Fig. 4, the $\mathbf{A_{ik}}$'s represent matrix sub-blocks and all the non-zero elements in the matrix appear only inside these sub-blocks. The blocks $\mathbf{A_{ii}}$, $\mathbf{A_{in}}$ and $\mathbf{A_{ni}}$ are said to form a *3-block group*, where $i \in [1, n-1]$ and $n \leq N$. $\mathbf{A_{nn}}$ is known as the last block. The $\mathbf{A_{ii}}$'s will be referred to as the diagonal blocks, and $\mathbf{A_{in}}$ and $\mathbf{A_{nk}}$ will be called right border block and bottom border block, respectively, where $i, k \in [1, n]$.

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

Figure 4. Sparse BDB matrix

The sparse BDB matrix format presents great advantages for parallel implementation. Since all non-border off-diagonal blocks contain only 0's, there will be no fill-ins in these blocks during factorization and the resulting matrix keeps the BDB structure of Fig. 4. In this matrix form, there is no data dependence among the factorization of the 3-block groups until the last block, so the factorization of the 3-block groups can be carried out independently from each other; no inter-processor communication is required during this procedure. The last block, $\mathbf{A_{nn}}$, requires data produced in the right and bottom border blocks, so its factorization is the last step. To factor the last block, pairs of blocks are multiplied in parallel to produce $A^*_{nn} = A_{nj}A_{jn}$, for $j \in [1, n-1]$. The summation of these products is needed to factor the last block. This summation is carried out along the binary tree in parallel by the other processors and the results are sent to the processor that was assigned the last diagonal block (for a highly parallel approach). We have achieved very good speedups [14] on our FPGA-based multiprocessor that targets an Altera SOPC FPGA board. We have also demonstrated our parallel BDB solution for Newton's load flow algorithm applied to power electric networks [15]. Although we observed that the factorization of the last block is the dominant sequential factor in speedups, we did not parallelize it in [14] because the last block is normally a dense block and frequent communications make the parallel implementation very inefficient. However, with our improved architecture in Fig. 1, we can employ the three neighbors that share the same data

memory in one sub-tree to factor the last block with much less overhead.

4. Load balancing

We need a good load balancing technique to translate the hardware parallelism into high speedups on real applications. Parallel LU factorization of sparse matrices is one of the hardest problems for load balancing due to its large amount of data dependences and occurrences of fill-ins. By ordering the matrices into BDB form, we eliminate the data dependences and communication during the factorization of the 3-block groups; this is the most time-consuming step. The unpredictability in execution times arises from the variant size of the 3-block groups and the number of fill-ins. Therefore, a dynamic load balancing strategy is needed to reduce the idle time between tasks and the worst-case execution time. Configurable logic allows us to modify the hardware design at any time in order to facilitate software optimizations. Since our system does not currently have operating system support, we built a dedicated system controller to take care of load balancing at runtime; in this approach, all processors must report their load information to the controller. This choice also minimizes the scheduling overhead, which is often a major disadvantage of dynamic load balancing.

4.1 Dynamic load balancing

Let us begin with the preprocessing phase where we attempt to order the matrix into an optimal BDB matrix; we do not consider the target hardware architecture at this time. The best ordering is the one that keeps the 3-block groups as dense as possible while not making the last block too large. This way we can reduce the number of floating-point operations in the procedure that follows. The best solution is also application-dependent. The partitioned BDB matrix is downloaded into the on-board SSRAM by the host. The information that the host computer passes to the SC includes at least the size of the matrix N, the number of diagonal blocks n, the size of the last block $n_n \times n_n$, the size of the diagonal block in every 3-block groups $n_i \times n_i$ (the sizes of the two border blocks in every 3-block group are determined by n_i and n_n) and its beginning memory address in the on-board memory.

Dynamic load balancing is carried out by the SC. There are three classes of tasks: factorization, multiplication and addition; they are implemented in assembly code and stored in the local program memory of every PE. The general task format is: **fac**/**mul**/**add**{n_i, n_n, #xxx}, where #xxx is the starting address of the 3-block group. During the system configuration phase, the SC assigns the initial loads based on the above information sent by the host computer. The SC keeps a load index for every processor in order to manage the unfinished tasks dynamically. A processor each time receives a 3-block group. The record entries include: the starting time of the working matrix group, the expected end time, the size of the working group, the possible next group for this processor, the phase the algorithm is in (that is, factorization or multiplication phase [14]) and finished groups. Based on the load index of PEs, the SC pre-fetches the data into the appropriate processor before it completes its current task. The candidate for the next task is the one with the lowest busy count, i.e. the percentage of the remaining work over the total working task. The SC always assigns the biggest available 3-block group in the task queue to the hungry PE. If the total number of the remaining tasks is less than the number of the PEs, then the SC tries to distribute the tasks to the PEs under different parents.

If a processor is idle and the task queue is empty, the SC first checks the status of the processors along the summation tree to find the nearest busy processor. If the idle processor is one of the two direct neighbors of a busy processor, then the SC modifies the ongoing task of the working processor and the idle processor is asked to share the work immediately via the shared memory without any data transfer. If it is not, the SC further decides whether it is worth asking the idle processor to help the working processor. The decision is based on the distance of the two processors, the size of the working group, how much work has already been done (i.e., the used time divided by the expected time), and the type of the current task. If it is during factorization, the idle processor will multiply the border blocks following factorization in the working processor. If the working processor is in the multiplication phase and the remaining work is greater than 1/3 (this number is based on the computation/communication time ratio on our machine), then the SC will copy half of the remaining data to the idle processor and modify the working processor's load information. The multiplication results will be collected along the binary tree. We tested our parallel BDB LU factorization with this dynamic scheduling policy for the standardized IEEE electric power 56-, 118- and 300-bus matrices for up to 7-processor systems on the Altera SOPC development board. Performance results are presented in Section 5.

4.2 Theoretical performance analysis

From the above discussion, we know that there are three basic operations in parallel BDB LU factorization: LU factorization of 3-block groups, multiplication of border blocks and addition of partial sums for submatrices of the same size as the last diagonal block. We first get the execution times of the three operations.

4.2.1. Execution times of basic operations. We assume that the three floating-point operations (+, - and *) take the same amount of time, T_f, and the floating-point division takes three times longer time, that is $3T_f$. These assumptions are reasonable for advanced floating-point units. We can show that the total time required for LU factorization of an n x n dense matrix is:

$$T_{lu}(n) = (\frac{4n^3 + 9n^2 - 13n}{6}) * T_f \quad (1)$$

So, the total factorization time for a 3-block group is a function of n_i and n_n, and is given by

$$T_{lu3}(n_i, n_n) = T_{lu}(n_i + n_n) - T_{lu}(n_i) \quad (2)$$

where $T_{lu}(n_i + n_n)$ and $T_{lu}(n_i)$ are evaluated by Eq. (1).

The total computation time for the multiplication of two matrices, $\boldsymbol{C} = \boldsymbol{A} * \boldsymbol{B}$, where $\boldsymbol{A}$ is a n_n x n_i matrix, $\boldsymbol{B}$ is an n_i x n_n matrix and $\boldsymbol{C}$ is an n_n x n_n matrix, is:

$$T_{mul}(n_i, n_n) = 2n_n^2 * n_i * T_f \quad (3)$$

The time required to add two n_n x n_n matrices is:

$$T_{add} = n_n^2 * T_f \quad (4)$$

For on-chip memory, every access takes only one clock cycle (assuming there is no contention for the same location). Let T_{clk} be the clock period; then, reading/writing a matrix of size n_n x n_i or n_i x n_n requires:

$$T_{mem}(n_i, n_n) = K * n_i * n_n * T_{clk} \quad (5)$$

where K is a constant associated with the processor and its shared memory. For our system, the average time for the transfer between the on-chip memory and the on-board SSRAM is $4T_{mem}(n_i, n_n)$.

4.2.2. Sequential execution time. If we assign all the BDB matrix blocks to a single processor, then the *N-1* independent 3-block groups will be processed in time:

$$\sum_{i=1}^{N-1} [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + 2T_{mem}(n_i, n_n)] \quad (6)$$

After the factorizations and multiplications, the processor will sum up all the partial results in time:

$$(N-2)[T_{add}(n_n) + T_{mem}(n_n, n_n)] \quad (7)$$

Finally, the remaining work is the factorization of the last block in time $T_{lu}(n_n)$ given by Eq. (1). To summarize, taking into account the startup time, T_{start}, used by the host to send the matrix and application code to the memory, and the close time, T_{end}, used to collect the factored data, the total execution time for one processor to factor the entire BDB matrix sequentially is computed as:

$$T_{seq} = T_{start} + \sum_{i=1}^{N-1} [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + 2T_{mem}(n_i, n_n)] + (N-2)[T_{add}(n_n) + T_{mem}(n_n, n_n)] + T_{lu}(n_n) + T_{end} \quad (8)$$

4.2.3. Parallel solution with static load balancing. For a parallel solution, the worst case for the speedup is when all p processors finish the work on *N-2* independent 3-block groups at the same time and only one 3-block group is left (the smallest block according to our scheduling policy). Then, the time spent on the *N-2* 3-block groups is:

$$\frac{1}{p} * \{\sum_{i=1}^{N-2} [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + 2T_{mem}(n_i, n_n)] + (N-2) * (T_{add}(n_n) + 2T_{mem}(n_n, n_n))\} \quad (9)$$

The last 3-block group will be handled by a single processor in time:

$$[T_{lu3}(n_{n-1}, n_n) + T_{mul}(n_{n-1}, n_n) + 2T_{mem}(n_{n-1}, n_n)] \quad (10)$$

After all the 3-block groups are finished, every node adds the partial sums of its two children and writes the result back to the memory, which will be accessed by its parent in the next step. The collection of the partial sums along the binary tree of depth $log_2(p+1)$ takes time:

$$\sum_{i=1}^{\log_2(p+1)} [2T_{add}(n_n) + 3T_{mem}(n_n)] \quad (11)$$

The last block is factored by the three neighbors after the summation of the partial sums. Since we use the shared-data memory between the three neighbors, we save on computation time but not on communication time: $\frac{1}{3}T_{lu}(n_n) + 2T_{mem}(n_n)$.

Thus, the total time required for static scheduling is:

$$T_{static} = T_{start} + \frac{1}{p} * \{\sum_{i=1}^{N-2} [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + 2T_{mem}(n_i, n_n)] + (N-2) * (T_{add}(n_n) + 2T_{mem}(n_n, n_n))\} + [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + 2T_{mem}(n_i, n_n)] + \sum_{i=1}^{\log_2(p+1)} [2T_{add}(n_n) + 3T_{mem}(n_n)] + \{\frac{1}{3}T_{lu}(n_n) + 2T_{mem}(n_n)\} + T_{end} \quad (12)$$

4.2.4. Parallel solution with dynamic load balancing. If we employ the proposed dynamic load balancing technique, then the work on the last 3-block group in the worst case will be performed by three neighboring processors (all the other p-3 processors are idle now). Eq. (10) is replaced by the following:

$$\frac{1}{3}[T_{lu3}(n_{n-1}, n_n) + Tmul(n_{n-1}, n_n)] + 2Tmem(n_{n-1}, n_n) \quad (13)$$

So, the total time is reduced to:

$$T_{dynmic} = T_{start} + \frac{1}{p} * \{\sum_{i=1}^{N-2} [T_{lu3}(n_i, n_n) + T_{mul}(n_i, n_n) + 2T_{mem}(n_i, n_n)] + (N-2) * (T_{add}(n_n) + 2T_{mem}(n_n, n_n))\} + \frac{1}{3}[T_{lu3}(n_{n-1}, n_n) + T_{mul}(n_{n-1}, n_n)] + 2T_{mem}(n_{n-1}, n_n) + \sum_{i=1}^{\log_2(p+1)} [2T_{add}(n_n) + 3T_{mem}(n_n)] + T_{lu}(n_n) + T_{end} \quad (14)$$

The upper bound on the speedup, $\frac{T_{seq}}{T_{dynmic}}$, is a complex function of $\{n_i, n_n, p, T_f, T_{clk}\}$. Since the factorization and multiplication have complexity $O(n^3)$, this algorithm will have good performance with a large number of

processors, as shown in Eq. (9). In the BDB matrix, the last block is usually larger than the independent diagonal blocks. The factorization of the last block is the dominant sequential limiting factor for the speedup. The size of the last block also imposes a big impact on the factorization and multiplication times, as demonstrated by Eqs. (2) and (5). We should try to make the last block as small as possible. On the other hand, the minimum size of the last block is largely determined by the physical characteristics of the original matrix. With more independent 3-block groups, we may have fewer floating-point operations (because of dynamic fillins within every 3-block group, the total time to factor all 3-block groups is not necessarily smaller for more 3-block groups) and get more benefits from fine-grain dynamic load balancing, but it also increases the time for summation of the partial results and the communication time.

5. Implementation and performance

5.1 Experimental results

In this experiment, we used the Altera SOPC development board. This board is populated with an APEX20KE FPGA device, the EP20K1500EBC652-1x that has 51,840 logic elements and 442,368 bits of on-chip memory. The board also contains two banks of SSRAM chips with a total size of 2 MB. We used a better floating-point multiplier than the one described in [14]; it requires only 880 logic elements. Thus, we implemented 7 computation processors plus the SC. The on-chip program memory for each PE is 4KB and the shared-data memory is 6KB. Because all the application code fits into the on-chip memory of each PE, we did not include caches for each PE in this experiment. All the programs are implemented in assembly code and are stored entirely in the on-chip program memory. The test matrices are the nodal admittance matrices of IEEE power flow test cases with 57, 118 and 300 buses, respectively (these numbers are the dimensionalities of the corresponding square matrices). Table 1 shows the partitioned results for these matrices due to the preprocessing phase.

Table 1. Partitioned IEEE power flow 57-, 118- and 300-bus test matrices

N	57	118	300
% of non-zeros	6.56	3.42	1.24
Dimensionality of the BDB diagonal blocks (n_i)	6,6,6,6,5, 4,4,4,4,12*	10,9,10,10, 9,9,10,10, 10,8,5,18*	18,12,18,16, 18,17,17,10, 14,14,18,18, 10,16,17,11, 17,8,31*

*: Last block

The speedups for the test matrices with up to 7 processors are shown in Table 2. We show in Fig. 5 the percentage of time needed to work on the last block compared to the total execution time in the 300-bus case. The transfer of matrix blocks between the on-board and on-chip memories becomes a bottleneck for a large number of PEs. We employed pre-fetching for data transfers and the improvement in performance for the 300-bus case is shown in Fig. 6.

5.2 Discussion

Generally, the parallel BDB factorization algorithm performs better with larger test cases for both the static and dynamic techniques, as shown in Table 2. This is because we have more jobs whose variations can smooth out the load imbalance. An exception is the 7-PE system for the 118-bus case. This should correspond to the worst case, which we discussed in Section 4.2. From Table 2, we can also see that the performance of dynamic load balancing is application (matrix) dependent. For example, it brought about more than 20% improvement for the 118-bus case with 7 PEs. For the 57-bus case, the systems with 5, 6 and 7 PEs got the similar speedups due to insufficient work for the PEs. Static scheduling performance for the 4-PE system is close to the worst case we analyzed in Section 4.2.3.; only one PE works on the last 3-block group while others are idle. Dynamic scheduling produced the biggest improvement compared to other scheduling policies for the 57-bus case, as expected by the analysis. Adding just one processor in the current scheduling policy does not improve the speedup. For the 118-bus and 300-bus cases, the largest improvement happens with the 7-PE system. It is the result of variations in the job size.

For most large-scale applications, such as power applications and circuit simulations, with increases in the matrix size, normally the sparsity decreases, which favors more diagonal blocks in the partitioned BDB matrices. However, the size of the last block increases as a result of more independent 3-block groups; its factorization could diminish the speedup due to more blocks. So, the bottleneck in parallel BDB factorization is the factorization of the last block. Since currently we use only three PEs to factor the last block based on the consideration of large communication latency for more PEs, the time spent on the last block becomes more significant with more PEs, as Fig. 5 shows.

6. Conclusions

New generation FPGAs provide tremendous design opportunities along several dimensions: system, hardware and software. We have presented a new shared-memory multiprocessor design and implementation on an Altera

SOPC board. Our target application is the LU factorization of sparse matrices. The factorization of the last block in the parallel LU factorization of sparse BDB matrices is a dominant sequential factor and we should try to minimize the size of the last block during the preprocessing phase; it was implemented here efficiently in parallel by taking advantage of the low communication overhead provided in our new architecture. We also introduced a new dynamic load balancing technique to minimize the impact of task unpredictability at runtime and improve performance. Our proposed dynamic load balancing algorithm was tested with the IEEE power flow 57-, 118- and 300-bus systems.

Table 2. Speedups (over the uni-processor) with static(S) and dynamic(D) load balancing for the IEEE test matrices

Number of PEs		2	3	4	5	6	7
57-bus	**S**	1.81	2.83	3.12	4.28	4.28	4.28
	D	1.90	2.90	3.51	4.30	4.87	4.87
118-bus	**S**	1.87	2.72	3.64	4.42	5.18	5.18
	D	1.92	2.85	3.72	4.73	5.21	6.29
300-bus	**S**	1.91	2.85	3.79	4.69	5.45	5.50
	D	1.95	2.89	3.84	4.68	5.65	6.21

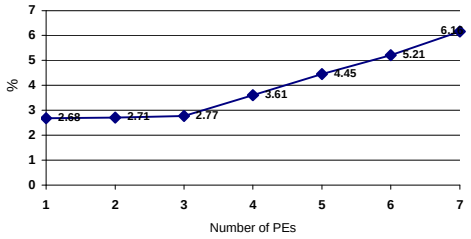


Figure 5. Percentage of the total execution time for the factorization of the last block in the 300-bus case

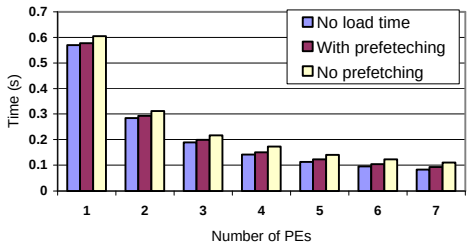


Figure 6. Execution times for the 300-bus test system

References

[1] W. A. Najjar, W. Bohm, B. A. Draper, J. Hammes, R. Rinker, J. R. Beveridge, M. Chawathe and C. Ross, "High-Level Language Abstraction for Reconfigurable Computing," *IEEE Computer*, Aug. 2003, pp. 63- 69.

[2] R. Hartenstein, "The Microprocessor is no more General Purpose: Why Future Reconfigurable Platforms Will Win," invited paper, *Int'l Conf. Innovative Syst. Silicon (ISIS'97)*, Austin, Texas, Oct. 1997.

[3] K. Sarrigeorgidis and J. M. Rabaey, "Massively Parallel Wireless Reconfigurable Processor Architecture and Programming," *10th Reconf. Archi. Works. (RAW 2003)*, Nice, France, April 2003.

[4] R. Hartenstein, "Are We Ready for the Breakthrough?" keynote address, *10th Reconf. Archi. Works. 2003 (RAW 2003)*, Nice, France, April 2003.

[5] B. Radunovic, "An Overview of Advances in Reconfigurable Computing Systems," *32nd Annual Hawaii Int'l Conf. System Sciences*, Maui, Hawaii, Jan. 1999.

[6] J. Frigo, M. Gokhale, and D. Lavenier, "Evaluation of the Streams-C C-to-FPGA Compiler: An Applications Perspective," *9th ACM/SIGDA Int'l Symp. Field Program. Gate Arrays*, Monterey, California, Febr. 2001, pp. 134-140.

[7] R. Hartenstein, "Trends in Reconfigurable Logic and Reconfigurable Computing," *9th Int'l Conf. Electronics Circuits Systems*, Vol. 2, 2002.

[8] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective," *IEEE Int'l Conf. Exhib. Design Autom. Testing Europe*, Munich, Germany, 2001, pp. 135-143.

[9] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software," *ACM Comput. Surveys*, Vol. 34, No. 2., June 2002, pp. 171-210.

[10] C. Wolinski, M. Gokhale and K. McCabe, "A Polymorphous Computing Fabric," *IEEE Micro*, Vol. 22, No. 5, Sep/Oct. 2002, pp. 56-68.

[11] J. Greenbaum, "Reconfigurable Logic in SoC Systems," *IEEE 2002 Custom Integrated Circuits Conf.*, 2002, pp. 5 –8.

[12] A. Sangiovanni-Vincentelli, L. K. Chen and L. O. Chua, "Node-Tearing Nodal Analysis," Tech. Report ERL-M582, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, Oct. 1976.

[13] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford Univ. Press, Oxford, England, 1990.

[14] X. Wang and S.G. Ziavras, "Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines," *Concurrency Computation: Prac. Expei.*, to appear in Vol. 16, No. 4, April 2004.

[15] X. Wang and S.G. Ziavras, "Parallel Solution of Newton's Power Flow Equations on Configurable Chips," *IEEE Trans. Power Systems*, submitted.

[16] X. Wang and S.G. Ziavras, "Parallel Direct Solution of Linear Equations on FPGA-Based Machines," *Works. Paral. Distrib. Real-Time Systems (with IPDPS2003)*, Nice, France, April 22-26, 2003, pp.113-120.