Reducing Energy Consumption of Parallel Sparse Matrix Applications Through Integrated Link/CPU Voltage Scaling

Seung Woo Son, Konrad Malkowski, Guilin Chen, Mahmut Kandemir, Padma Raghavan
The Pennsylvania State University
Department of Computer Science and Engineering
University Park, PA 16802 USA
{sson,malkowsk,guilchen,kandemir,raghavan}@cse.psu.edu

Abstract

Reducing power consumption is quickly becoming a first-class optimization metric for many high-performance parallel computing platforms. One of the techniques employed by many prior proposals along this direction is voltage scaling and past research used it on different components such as networks, CPUs, and memories. In contrast to most of the existent efforts on voltage scaling that target a single component (CPU, network or memory components), this paper proposes and experimentally evaluates a voltage/frequency scaling algorithm that considers CPU and communication links in a mesh network at the same time. More specifically, it scales voltages/frequencies of CPUs in the nodes and the communication links among them in a coordinated fashion (instead of one after another) such that energy savings are maximized without impacting execution time. Our experiments with several tree-based sparse matrix computations reveal that the proposed integrated voltage scaling approach is very effective in practice and brings 13% and 17% energy savings over the pure CPU and pure communication link voltage scaling schemes, respectively. The results also show that our savings are consistent with the different network sizes and different sets of voltage/frequency levels.

Keywords: Energy consumption, dynamic voltage scaling, parallel sparse matrix, computation, communication networks.

1 Introduction

Power consumption is becoming a critical issue for high-end computing platforms due to several factors including cost, space, reliability, and maintenance. Consequently, recent research efforts from different groups in both academia and industry have focused on techniques that help us accurately model and reduce power consumption of different hardware components in a large computing infrastructure. These studies, details of which are discussed in Section 2, include CPU power optimizations, memory banking and low-power operating mode management, network power minimization, and energy-oriented disk I/O optimizations.

Voltage and frequency scaling has been identified by past research as one of the most effective ways of reducing CPU power [10, 34]. More recently, there have been proposals [30, 35] that apply voltage/frequency scaling to network links to save communication power. However, to our knowledge, none of the prior efforts in the domain of high-performance computing considered using voltage scaling on both CPUs and communication links of a given parallel architecture in a coordinated fashion to save power. The work described in this paper is a step in this direction. More specifically, focusing on sparse matrix computations that can be represented as trees, this paper studies the potential benefits that can be accrued when using CPU and communication link voltage/frequency scaling in a coordinated fashion. To achieve this, we propose and experimentally evaluate a voltage/frequency scaling algorithm.

In this paper, we propose a *voltage scaling* based energy reduction scheme for tree-based parallel sparse computations. Our first approach is based on extracting a representation of load imbalances across the processors in the parallel system, and using this information in assigning the most suitable supply voltages and

frequencies to processors in the system. This representation is extracted after applying the load-balancing techniques available for the problem [12, 20, 26, 27]. We note that many state-of-the-art processors (e.g., [1, 33, 19]) employ circuit mechanisms that support voltage/frequency scaling; and in large parallel systems built from such components, the voltage/frequency of each processor can be scaled independently of others. Our goal is to reduce the energy consumption of processors through voltage/frequency scaling as much as possible, without increasing the execution time of the application. Therefore, our approach exploits load imbalance across parallel processors, and applies voltage scaling to only the processors that are not in the critical path.

Based on our voltage/frequency scaling techniques targeting CPU power, we next propose and evaluate our integrated technique that scales down the voltages of both CPUs and communication links. An important characteristic of the proposed algorithm is that it scales the voltages of CPUs and links considering the impact of doing so on each other; this is radically different from an alternate approach that applies CPU voltage scaling after communication link voltage scaling or vice versa. To test the effectiveness of our approach, we applied it to a set of tree-based sparse matrix computations running on a two-dimensional mesh network and compared it to two alternate schemes, one that applies voltage scaling only to CPUs and the other one that applies voltage scaling to only communication links. Our experiments reveal that the proposed integrated voltage/frequency scaling approach is very effective in practice and brings 13% and 17% energy savings over the pure CPU and pure communication link voltage scaling schemes. The results also show that our savings are consistent with the different network sizes and different sets of voltage/frequency levels.

The remainder of this paper is structured as follows. In the following section, we describe the related work on voltage scaling in the context of the interconnection networks and processors. Section 3 explains the tree based computation model for parallel sparse matrix solvers. In Section 4, we propose several voltage scaling techniques for tree based parallel matrix solvers, and present our evaluation methodology and results with our algorithms given in the same section. Our integrated link/CPU voltage scaling algorithm is presented and experimentally evaluated in Section 5. We conclude the paper in Section 6 with a summary of our major contributions.

2 Related Work

Several studies in the past have proposed dynamic voltage scaling (DVS) techniques for reducing energy consumption of communication links in the NoC (Network-on-Chip) based systems and high-end multiprocessor systems [30, 31, 35]. The main idea behind these approaches is to scale down the voltage/frequency of communication links when there is enough communication slack (i.e., the amount of latency by which communication can be delayed without affecting overall execution time) observed or predicted. In order for these DVS techniques to be feasible, Kim et al [22] proposed serial links that can operate under various link voltage/frequency levels. Employing links with variable voltage/frequency, Shang et al [30] presented and evaluated a history-based DVS scheme for the communication links. Worm et al [35] proposed an adaptive low-power transmission technique for on-chip networks, whereas Shin et al [31] discussed a task mapping technique based on genetic algorithms to utilize voltage scalable links for saving energy in NoC based systems. Besides DVS techniques for communication links, several techniques that shut down unused or underutilized links have been proposed. Kim et al [21] proposed a dynamic link shutdown (DLS) technique for chip-to-chip networks. Soteriou et al [32] explored the design space for communication links with turn on/off capability.

In addition to these efforts that target reducing power consumption in communication links, there are also studies that target reducing power/energy consumption of large server and cluster systems. These efforts can be broadly classified into three categories. The first category of the efforts considered CPU-centric techniques that turn off unused CPUs [8] or scale down CPUs that execute non-critical execution [9, 10, 4]. Voltage

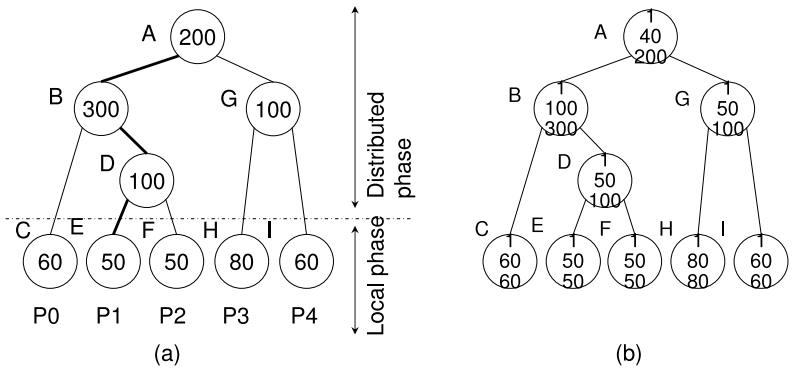


Figure 1: An example weighted tree and its VTE (Voltage-Time-Energy) tree. (a) Weighted tree. The numbers written inside the nodes indicate the associated computational cost. Leaf nodes represent the local phase computations, and each of them is assigned to a processor (P0-P4). The bold lines represent the critical path. (b) VTE tree. The three numbers inside each node, from top to bottom, represent the voltage level, the time it takes, and the energy consumption required to compute this node.

scaling on processors [34] has been extensively studied and several commercial processors (e.g., Transmeta's Crusoe [33] and AMD's Athlon 64 [1]) already provide mechanisms to control the frequency and voltage of processors. The second category of studies [3, 21, 5] proposed several techniques that focus on individual components of the server based computing systems such as CPUs and main memory. Lastly, many studies focused on reducing energy consumption on the disk subsystem, which is a huge energy consumer for large data centers, by completely spinning down disks [7] or dynamically adjusting the rotational speed of disks [16]. As compared to these prior efforts, our approach combines CPU and communication link voltage scaling.

Lastly, in the domain of real-time distributed embedded systems, Luo et al [23] proposed a technique that simultaneously scales voltages of processors and communication links. Our approach is different from Luo et al's work in that we focus exclusively on parallel sparse matrix applications and consider the underlying network topology in selecting proper link voltages (and corresponding frequencies). Consequently, our integrated voltage scaling algorithm is entirely different from theirs.

3 Tree-Based Computation Model

In this paper, we concentrate on parallel sparse linear systems to study the impact of CPU and link voltage scaling without impacting performance. Such computations typically dominate the execution time of many large-scale parallel applications on multiprocessors and clusters of workstations. There are many classes of parallel sparse linear solvers and important classes include parallel direct solvers based on sparse factorization [6, 14, 20, 24], iterative solvers [18, 29], and direct-iterative hybrids through preconditioning [15, 25, 28]. While there is no single method that is always better than others across the different application domains and the underlying parallel execution platforms, they share the same notion that a given sparse matrix can be represented as a graph, which can be partitioned across processors for parallel execution. This partitioning [20] is usually performed using a recursive scheme for computing vertex or edge separators, and the associated partitioning tree (and related trees) can serve as a useful model for the underlying parallelism and data dependencies.

We focus on tree-based parallel sparse computations that are representative of parallel sparse solvers when matrix is symmetric positive definite. Such solvers consist of a initial symbolic phase followed by a numeric phase. In the symbolic phase, the matrix for parallel computation is partitioned to determine the actual structure of the Cholesky factor L [17]. The numeric phase, which represents the dominant cost in total solver time,

involves computation of the sparse factor and solving the problem using the determined sparse factor. The columns of L can be clustered into *supernodes*, each of which contains a set of consecutive columns with the same zero/nonzero structure. The overall numeric phase can be performed in parallel on *tree of supernodes.* The tree structure represents the data dependencies, and each tree node denotes a supernode of L and its corresponding set of dense-matrix operations. The allocation of processors to subtrees is based on the weights on the tree to represent the computation cost. While the allocation procedure can be done in several phases to balance computational load on each processor, inherent irregularities in the sparse matrix often result in workload imbalance across processors during parallel sparse matrix computation.

In general, for sparse systems from modeling and simulation applications with coefficient matrices of dimension N, the number of levels in the tree is approximately $log_2 N$. For illustrative purposes, let us assume that the tree has more leaves than the number of processors P. More often, the top 5 to 10 highest levels of the tree nodes account for more than 50% of the total communication and computation time [24]. Since process execution roughly occurs at $log_2 P$ levels from the root and the subtrees rooted below these nodes are assigned to the P different processors (as local computations at processors), hence the tree based sparse system with the number of P processors is scalable because it is applied in detail to the top $log_2 P$ levels.

In this paper, we use such weighted trees as the model of computation. Specifically, the weighted paths in this tree can be used to compute loads at each processor, and to determine the *critical path* (corresponding to the largest load across all processors). An example weighted tree is depicted in Figure 1(a). In this figure, the number inside each node represents the computational cost (load) at that node, and we use capital letters (A-I, in this example) to identify different nodes. Leaf nodes represent the local phase computations, and each of them is assigned to a single processor. For example, node C is assigned to processor P0, and node E is assigned to processor P1. Root nodes of different tree/subtrees represent the distributed phase computations. The computations at a root node of a tree/subtree are distributed evenly across all the processors to which the leaf nodes of this tree/subtree are assigned. For example, the computation in node D in Figure 1(a) is assigned to processors P1 and P2, with each processor having 50 units of computational cost for processing node D. Similarly, the computational load represented by node A is assigned to all the five processors, with each of them having 40 ($= 200/5$) units of computational cost. It should be noted that the processors sharing a node's computations need to synchronize with each other before they start the computations at this node. For example, although P2 could finish the computations at node F before P1 could finish its computations at node E, P2 must wait until P1 finishes before both the processors could co-operate to start the computations at node D. In such a weighted tree structure, the processor with the largest load (when considering all the nodes it is involved with) determines the critical path. In Figure 1(a), P1 has the largest load, and the critical path is highlighted using bold lines. When there is no confusion, we use the root node of a tree/subtree to represent that tree/subtree. For example, 'subtree B' refers to the subtree consisting of node B, node C, node D, node E, and node F. Given a weighted tree, our approach tries to maximize energy savings during its execution.

4 CPU Voltage Scaling

4.1 Algorithms

Over the range of allowed supply voltages, the highest frequency at which a CPU can run correctly drops proportionally to the supply voltage (i.e., $f \propto V$). Since the main component of power consumption is proportional to $V^2 f$, it is easy to see that reducing V has a quadratic effect on energy consumption. Consequently, a CPU can save substantial energy by running with lower supply voltage (hence, more slowly) [2]; e.g., by reducing its supply voltage to half, it can reduce its energy consumption to $1/4th$ of the original. The important point to note here is that, for correct operation, when voltage is scaled, frequency needs also be scaled.

Table 1: Voltage/frequency/power levels used in our examples.

Voltage	Frequency	Power
1	1	1
0.8	0.8	0.512
0.6	0.6	0.216
0.4	0.4	0.064

```
VoltageScaling(node)
{
    if (node.hasChildren) {
        AdjustTreeVoltage(node.left, node.level);
        AdjustTreeVoltage(node.right, node.level);
        if (node.left.treeTime < node.right.treeTime) {
            fastNode = node.left;
            slowNode = node.right;
        }
        else {
            fastNode = node.right;
            slowNode = node.left;
        }
        for (newLevel = fastNode.level + 1;
                newLevel < MAXLEVEL; newLevel++)
            if (TreeTimeAtLevel(fastNode, newLevel)
                    > slowNode.treeTime)
                break;
        newLevel = newLevel - 1;
        AdjustTreeVoltage(fastNode, newLevel);
        VoltageScaling(node.left);
        VoltageScaling(node.right);
    }
}
```

```
AdjustTreeVoltage(node, lev)
{
    node.level = lev;
    node.nodeTime = NodeTimeAtLevel(node, lev);
    node.treeTime = TreeTimeAtLevel(node, lev);
}

AdjustNodeVoltage(node, lev)
{
    childrenTime = node.treeTime - node.nodeTime;
    node.level = lev;
    node.nodeTime = NodeTimeAtLevel(node, lev);
    node.treeTime = node.nodeTime + childrenTime;
}

TreeTimeAtLevel(node, lev)
{
    return (node.origTreeTime / FREQ[lev]);
}

NodeTimeAtLevel(node, lev)
{
    return (node.origNodeTime / FREQ[lev]);
}
```

Figure 2: VS1. VoltageScaling() is the main function, and the other four routines are helper functions. The complexity of this algorithm is $O(LN)$, where L is the number of available voltage levels, and N is the number of nodes in the tree.

In this section, we first present the algorithms for dynamically varying (scaling) CPU speed and voltage to save energy in tree-based parallel sparse computations. Given a tree, our main objective is to find a dynamic voltage scaling scheme that can maximize energy savings without affecting the overall original execution time (the execution time taken without any power management). We observe that the load imbalance in the tree can be utilized to reduce energy consumption. Specifically, for those nodes that are not in the critical path, their execution speed (frequency) can be reduced without affecting the overall execution time, and their energy consumption can be reduced via voltage scaling.

For the examples presented in this section, we assume the power numbers (levels) given in Table 1. All the numbers in this table are normalized, and the original voltage/frequency/power numbers are $1/1/1$. We use a *VTE* (*Voltage-Time-Energy*) *tree* to represent the voltage assignments for a weighted tree. Figure 1(b) gives an example of VTE tree corresponding to the weighted tree illustrated in Figure 1(a). The three numbers inside each node of a VTE tree, from top to bottom, correspond to the voltage level, time spent, and energy used to compute that node (note that, these numbers are also normalized). For example, the voltage level, time, energy consumption used to compute node G, are 1, 50, and 100, respectively. In other words, for node G, we assign voltage level 1 to processors P3 and P4, and the time spent and energy consumption incurred are 50 (each processor gets 50 units of computation, and they run in parallel) and 100 ($time * power * number\_of\_processors$), respectively.

A general rule that we follow in our algorithms is that it is more beneficial to scale a given weighted tree

as a whole, rather than to scale the nodes one after another. In other words, under the same performance loss bound (i.e., allowable performance degradation), a voltage scaling scheme that assigns similar voltage levels to different nodes in the tree should result in better energy savings than a scheme that assigns different levels to nodes in the tree. A simple example can help us explain why this rule makes sense. Suppose that we have two nodes which we need to run sequentially. Let us assume that the $(Voltage, Time, Energy)$ values of these two nodes are $(V, 2T, 2E)$ and (V, T, E). Assume further that the maximum allowable execution time is $6T$. If we scale only the first node to $(0.4V, 5T, 0.32E)$, the energy saving achieved would be $1.68E$. On the other hand, if we scale the two nodes to the same voltage $0.5V$, which means that their $(Voltage, Time, Energy)$ values become $(0.5V, 4T, 0.5E)$ and $(0.5V, 2T, 0.25E)$, the energy saving obtained would be $2.25E$. Therefore, in this example, the latter scheme, which scales the two nodes as a whole, generates better energy saving result. We can generalize this argument because, for the same node, the performance penalty to save a certain amount of energy is higher when the voltage level is lower (in fact, the performance penalty is a linear function of the inverse of CPU frequency).

Our first algorithm, called *VS1*, is a recursive one that follows the above rule. For the root node of the tree being considered, one of its children is in the critical path and cannot be scaled as a whole. For the other child that is not in the critical path, we can scale it and its descendants down together until we reach a point where more aggressive scaling will increase the overall original execution time of the tree. After that, we scale the two children recursively using this algorithm; i.e., we apply the same algorithm to the two children of the root, and so on. Figure 2 gives this algorithm in the pseudo-code format. VoltageScaling() is the main function and the other four routines are helper functions. Note that, in VoltageScaling(), we try to scale the faster subtree first, which is not in the critical path, as a whole (see the for-loop). Then, we scale the two children recursively by invoking VoltageScaling() for each of them. It can be observed from VS1 that each node is visited only once, and the for-loop is the only loop in the function. Assuming that there are L voltage levels available and N nodes in the tree, the complexity of VS1 is $O(LN)$. Note that the proposed algorithm, VS1, makes a "greedy" approach in a sense that, as it traverses each tree node, it tries to find an optimal voltage/frequency level at each node, which gives the best energy savings. Figure 3 illustrates how VS1 is applied to the weighted tree shown in Figure 1. Figure 3(a) is the original VTE tree, and the total energy consumption is 1000. Subtree G is scaled first since it is the faster child of node A (Figure 3(b)). We find that 0.8 is the lowest possible voltage level that we can assign to subtree G without making it slower than subtree B (see Table 1). After that, we scale subtree B, and node C is the faster child, which is assigned voltage level 0.6 (Figure 3(c)). Subtree D cannot be scaled any further since both its children take the same amount of time to finish. Next, we scale subtree G, and node I is its faster child (Figure 3(d)). Voltage level 0.6 is the lowest possible voltage for node I without making it slower than node H. Figure 3(d) depicts the final VTE tree after applying VS1, and the total energy consumption when employing these voltage assignments is 858.4, a 14.2% reduction compared to the original energy consumption.

It should be noted that VS1 generates the optimum voltage/frequency scaling scheme in terms of energy savings only if we have continuous voltage levels. However, in reality, we have only a limited set of voltage levels that can be used. Under such discrete voltage levels, VS1 is no longer the optimum choice. For example, in Figure 3(d), we can scale the voltage/frequency of node G down further as shown in Figure 4(a), or scale the voltage/frequency of node H as shown in Figure 4(b). We can observe here that, in some cases, even though it is not possible to scale down a tree as a whole further, it may be still possible to scale some individual nodes without hurting the performance (i.e., without exceeding the allowable performance degradation). In Figure 5, we give two possible options, *VS2* and *VS3*, which exploit such opportunities. To obtain voltage assignments for a tree rooted at node A, we call VoltageScaling(A, 0). Note that, in VS2 and VS3, under a given allowable performance loss (which can be 0 meaning that no performance loss is to be tolerated), we first try to scale the whole tree as much as possible (as in the case of VS1). When we reach the point where scaling the whole

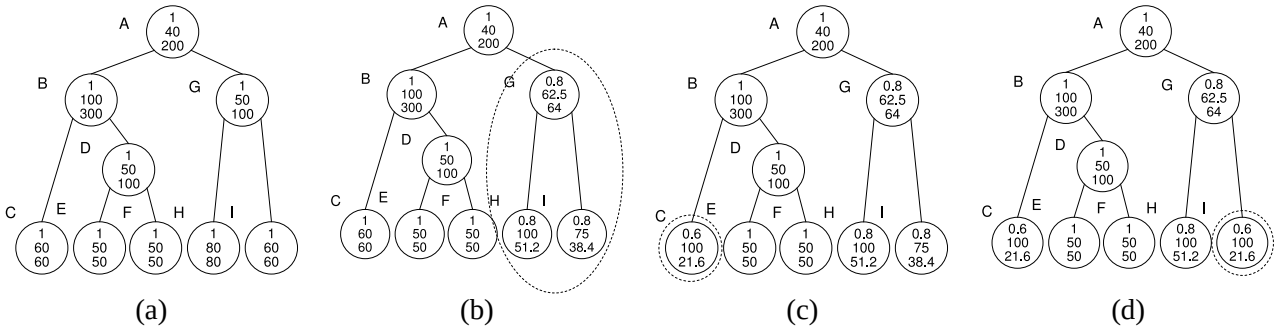


Figure 3: Example application of VS1. (a) The original VTE tree. (b)-(d) Different steps of applying VS1 to the tree. The subtree or node in dashed circle is the one being scaled in the corresponding step.

tree further will exceed the allowable performance loss, we begin to select some individual nodes in the tree as candidates for further voltage/frequency scaling. We have several choices in selecting such individual nodes. We can scale the root node first, then scale the two children, or we can scale the children first, and then scale the root node. We can even have an adaptive scheme that would make its decisions based on the weight distribution or other possible factors. In this work, we implement only the first two choices: *root-first* and *children-first*. Figure 5(a) gives an algorithm, called *VS2*, which is the root-first version. Figure 5(b) gives an algorithm, called *VS3*, which is the children-first version. A major difference between these two algorithms is in the order of scaling the root node (the for-loop) and scaling the children (two recursive calls). In Figure 5(a), the for-loop comes before the recursive calls, which means that we try to scale the root node first to exploit the slack available after scaling the whole tree. In comparison, in Figure 5(b), the two recursive calls are invoked before the execution of the for-loop, which means that we try to scale the children first to exploit the slack available after scaling the whole tree. In both VS2 and VS3, each node in the tree is visited only once, and there are two loops whose iteration counts are the number of available voltage levels. Assuming that there are L voltage levels available and N nodes in the tree, the complexities of both VS2 and VS3 are $O(LN)$. Figure 6 gives an example application of VS2 to the VTE tree shown in Figure 3(b). The scenarios shown in Figure 6(a) and Figure 6(b) are similar to those with VS1, as shown in Figure 3(b) and Figure 3(c). After we scale the subtree G as a whole, the execution time of subtree G (183.3) is still smaller than the execution time of subtree B (200). In VS1, we are not able to exploit this slack because we cannot scale subtree G as a whole further. But, using VS2, we can exploit the slack of subtree G by scaling its root node, G, to voltage level 0.6. After that, we scale its children, and find that node I can be scaled down further. It can be observed that, in this example, VS2 performs better than VS1 since it saves more energy in computing node G. Figure 7 gives an example application of VS3 to the VTE tree shown in Figure 3(b). The first two steps in Figure 7 are the same as those in Figure 6. The difference between VS2 and VS3 in this example is in the order of exploiting the slack due to subtree G. In VS3 (Figure 7(c)), we exploit the slack by scaling the voltages/frequencies of two children first. After scaling the children, we cannot scale node G any further since there is no enough slack left for node G.

In the VoltageScaling() functions of both VS2 and VS3, the parameter *slack* indicates the maximum execution time increase (i.e., performance degradation/loss) allowed to save energy. Consequently, both VS2 and VS3 can work under a given performance degradation bound. Assuming that the original execution time of a tree R is T and the maximum percentage performance loss that can be tolerated is P, we can invoke VoltageScaling(R, $T * P$) to obtain voltage assignments under such a performance constraint. Note that, when we use VoltageScaling() with parameter 0, this indicates that we are not tolerating any increase in the original execution time. This is the strategy that we use in most of our experiments. VS1 can also be implemented in a

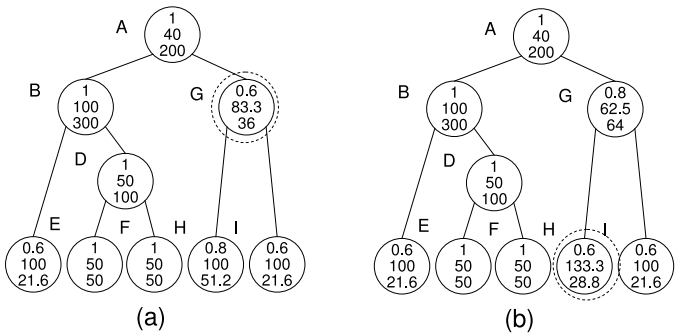


Figure 4: Two examples showing that the result achieved by VS1 in Figure 3(d) is not optimum. The nodes in dashed circles have been scaled further.

```
VoltageScaling(node, slack)
{
    maxTreeTime = node.treeTime + slack;
    for (newLevel = node.level + 1; newLevel < MAXLEVEL;
            newLevel++)
        if (TreeTimeAtLevel(node, newLevel) > maxTreeTime)
            break;
    newLevel = newLevel - 1;
    AdjustTreeVoltage(node, newLevel);
    slack = maxTreeTime - node.treeTime;
    if (node.hasChildren) {
        choose some individual nodes from the tree for scaling.
    }
    if (node.hasChildren) {
        AdjustTreeVoltage(node.left, node.level);
        AdjustTreeVoltage(node.right, node.level);
        for (newLevel = node.level + 1;
                newLevel < MAXLEVEL; newLevel++)
            if (NodeTimeAtLevel(node, newLevel)
                    > node.nodeTime + slack)
                break;
        newLevel = newLevel - 1;
        AdjustNodeVoltage(node, newLevel);
        slack = maxTreeTime - node.treeTime;
        if (node.left.treeTime < node.right.treeTime) {
            fastNode = node.left;
            slowNode = node.right;
        }
        else {
            fastNode = node.right;
            slowNode = node.left;
        }
        extraSlack = slowNode.treeTime - fastNode.treeTime;
        VoltageScaling(slowNode, slack);
        VoltageScaling(fastNode, slack+extraSlack);
        if (slowNode.treeTime > fastNode.treeTime)
            node.treeTime = node.nodeTime + slowNode.treeTime;
        else
            node.treeTime = node.nodeTime + fastNode.treeTime;
    }
}
```

(a) VS2: Root-first version.

```
VoltageScaling(node, slack)
{
    maxTreeTime = node.treeTime + slack;
    for (newLevel = node.level + 1; newLevel < MAXLEVEL;
            newLevel++)
        if (TreeTimeAtLevel(node, newLevel) > maxTreeTime)
            break;
    newLevel = newLevel - 1;
    AdjustTreeVoltage(node, newLevel);
    slack = maxTreeTime - node.treeTime;
    if (node.hasChildren) {
        choose some individual nodes from the tree for scaling.
    }
    if (node.hasChildren) {
        AdjustTreeVoltage(node.left, node.level);
        AdjustTreeVoltage(node.right, node.level);
        if (node.left.treeTime < node.right.treeTime) {
            fastNode = node.left;
            slowNode = node.right;
        }
        else {
            fastNode = node.right;
            slowNode = node.left;
        }
        extraSlack = slowNode.treeTime - fastNode.treeTime;
        VoltageScaling(slowNode, slack);
        VoltageScaling(fastNode, slack+extraSlack);
        if (slowNode.treeTime > fastNode.treeTime)
            node.treeTime = node.nodeTime + slowNode.treeTime;
        else
            node.treeTime = node.nodeTime + fastNode.treeTime;
        slack = maxTreeTime - node.treeTime;
        for (newLevel = node.level + 1;
                newLevel < MAXLEVEL; newLevel++)
            if (NodeTimeAtLevel(node, newLevel)
                    > node.nodeTime + slack)
                break;
        newLevel = newLevel - 1;
        AdjustNodeVoltage(node, newLevel);
    }
}
```

(b) VS3: Children-first version.

Figure 5: Two versions of voltage/frequency scaling algorithm. The helper functions, including AdjustTree-Voltage(), AdjustNodeVoltage(), and NodeTimeAtLevel(), are defined in Figure 2. The complexity of both the versions are $O(LN)$, where L is the number of voltage levels, and N is the number of nodes in the tree.

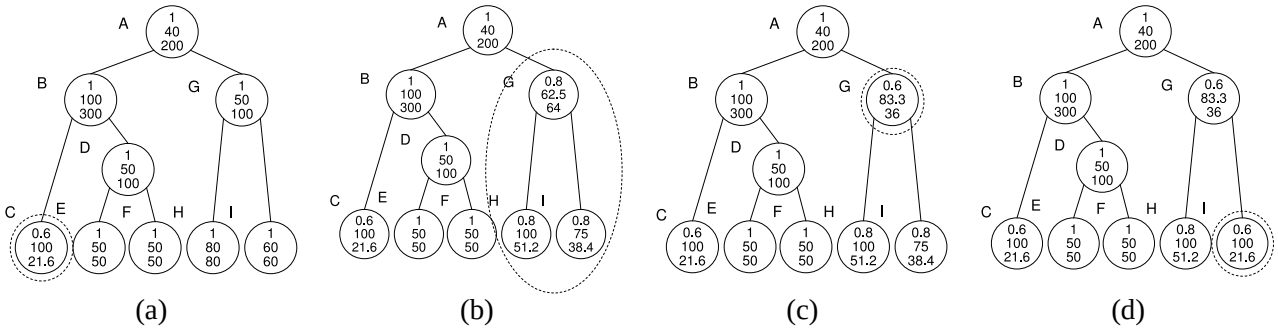


Figure 6: Example application of the VS2 algorithm (root-first version). The original VTE tree is given in Figure 1(b). (a)-(d) Different steps. The subtree or node in dashed circle is the one being scaled in the corresponding step.

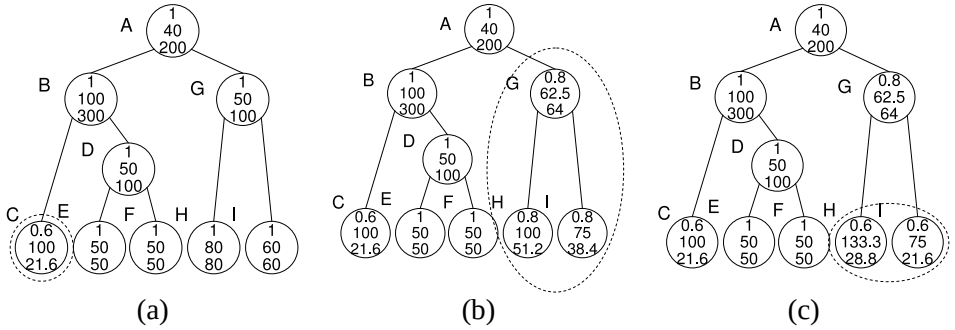


Figure 7: Example application of the VS3 algorithm (children-first version). The original VTE tree is given in Figure 1(b). (a)-(c) Different steps. The subtree or node in dashed circle is the one being scaled in the corresponding step.

slack-based fashion, and thus can also be used in cases where performance constraints are specified.

It is also to be mentioned that there is some cost for a processor to transition between different voltage/frequency levels. However, this cost is usually very small compared to the application execution time. In addition, the voltage/frequency transitions are not very frequent, and occur only across the levels in the tree. Although it is not shown explicitly in the algorithm presented, we have taken this cost into account in our implementations and experiments with all our schemes.

4.2 Experimental Results

4.2.1 Experimental Setup

To evaluate our approach, we implemented a simulation platform shown in Figure 8. We first obtain trace data, which indicate the computation weight involved at each level of the tree from parallel sparse matrix solver. This trace data is then fed to the energy simulator along with the CPU power models. Based on the given voltage scaling method, which was explained in the previous section, the energy simulator generates energy and performance statistics. Note that, the trace data for communication weight and link energy model will be used when we present our integrated algorithm that scales voltages of both CPU and link later in this paper. It should be emphasized that the voltage/frequency decision by each voltage scaling scheme is made statically based on the trace data we actually collected. Therefore, our evaluation does not consider any dynamic change

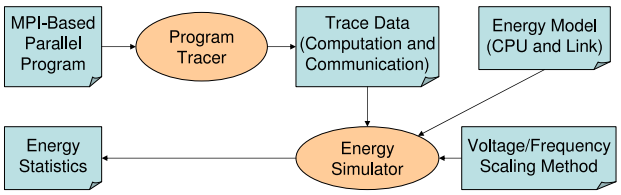


Figure 8: Our simulation platform.

Table 2: Default simulation parameters for CPU.

Parameter	Value
CPU clock frequency range	800MHz $\sim$ 2400MHz
Number of voltage/frequency levels	5
Supply voltage range	1.1 $\sim$ 1.5V
Default clock frequency	2400 MHz
Voltage/frequency transition latency	1 ms

Table 3: Our test-suite of sparse matrices from practical applications.

Matrix	**P**	**Rank**	$\|A\|(10^3)$	$\|L\|(10^3)$	**Factorization Cost** (10^6)	**Ideal** (10^6)	**Max** (10^6)	**Min** (10^6)	**Depth of Tree**
bmw7st1	64	141347	3741	81340	217764	3403	3831	2045	10
bcsstk31	28	35588	608	10605	7161	256	295	194	7
bcsstk35	17	30237	740	11335	11996	706	834	637	7
crystk02	11	13965	491	5059	2890	263	406	158	5
finan512	28	74752	335	17905	13695	490	739	376	8
nasasrb	22	54870	1366	13084	6193	282	444	196	7
tube1	7	21498	459	5731	3865	553	826	405	4

in the CPU's load.

We use the power numbers from AMD Athlon processor [1], and Table 2 gives the default simulation parameters including power and performance values. All processors operate at the highest frequency of 2400MHz by default (i.e., without any voltage/frequency scaling). In the rest of the paper, the energy consumption values presented are normalized with respect to the *default* (*base*) *version*, where *no* power saving scheme is employed. It should be noted that, all the experimental results presented in this paper already take into account the extra cost of processors' transitioning between different voltage/frequency levels.

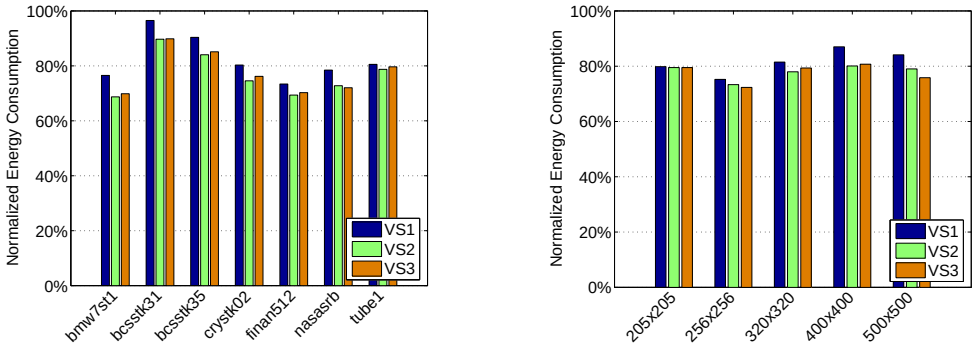
Our test suite is derived from parallel sparse direct solution using a multifrontal scheme [24] on several sparse matrices from practical applications described in Table 3. These sparse matrices tend to be irregular and are representative of matrices for a broad range of scientific computing applications. The second column of Table 3 gives the number of processors available for the applications. The third column shows the ranks of matrices. The fourth and the fifth columns give, respectively, the number of nonzeros in matrix A before factorization, and the number of nonzeros in factor L of matrix A after factorization. The sixth column shows the number of arithmetic operations required to factor matrix A into the form of LL^T. The seventh column gives the ideal workload per processor, which is obtained by dividing the total workload over the number of processors. The next two columns list the maximum and minimum workloads assigned to different processors. Finally, the last column presents the depth of tree when each application is represented as a tree computation model described in Section 3. In addition to those matrices that are extracted from real applications, we also use some *model* sparse matrices that are automatically generated to test the scaling of our approach as the problem size is increased with the number of processors (while maintaining the work per processor at a fixed level). The two-dimensional model problem is associated with the $K \times K$ five-point finite difference grid, and results in a sparse matrix of rank $N = K^2$ [11]. Table 4 gives the description of these problems. The meaning of the columns in this table is the same as those in Table 3.

4.2.2 Results

Figure 9 presents the normalized energy consumptions with our three different CPU voltage scaling algorithms (VS1, VS2, and VS3, explained in Section 4.1). All bars of a given solver are normalized with respect to the execution when *no* voltage/frequency scaling scheme is employed. Note that, unless otherwise stated, all

Table 4: Our test-suite of model sparse matrices.

Matrix	**P**	**Rank**	$\|A\|(10^3)$	$\|L\|(10^3)$	**Factorization Cost** (10^6)	**Ideal** (10^6)	**Max** (10^6)	**Min** (10^6)	**Depth of Tree**
205x205	3	42025	125	1143	103	35	50	25	3
256x256	7	65536	196	1913	204	30	46	22	4
320x320	15	102400	306	3139	398	27	37	21	5
400x400	31	160000	479	5191	799	26	34	19	7
500x500	63	250000	749	8607	1578	25	34	20	8



(a) Practical problems. (b) Model problems.

Figure 9: Normalized energy consumptions with different strategies.

our energy savings are achieved under no performance loss. It can be observed from this figure that our strategies save significant amount of energy by using voltage/frequency scaling. The average energy savings brought by VS1, VS2, and VS3 are 16%, 21%, and 21%, respectively, for the practical problems (see Table 3). The corresponding energy savings brought by the three schemes for model problems (see Table 4) are 17%, 21%, and 21% in the same order. Our schemes save less energy with bcsstk31; but, as we will discuss in Section 4.2.3, the main reason for this behavior is the inherent limited opportunity (more evenly distributed workload) in these two cases.

We observe that VS2 and VS3 are both better than VS1 across all the cases tested. Recall that, the difference between VS1 and the other two schemes is that VS1 always tries to scale a subtree as a whole, while VS2 and VS3 also scale individual nodes when scaling a whole subtree further is not possible due to the limited number of voltage levels available and the specified performance bound. The improvements brought by VS2 and VS3 over VS1 indicate that it is important to take into account the number of available voltage levels when applying voltage/frequency scaling to these problems. On the other hand, there is no clear winner between VS2 and VS3, and the energy savings brought by these two schemes are similar for all the cases tested. Unless otherwise stated, we use VS3 as the default scheme for most of the experiments presented in the rest of this paper.

So far, we required that the total execution time should not be affected by voltage/frequency scaling. That is, we try to achieve the maximum energy benefits that can be obtained without increasing the original execution time. In some execution environments, however, one might be willing to tolerate some performance penalty for larger energy savings. Figure 10 presents the results under such a scenario. The values on the x-axis represent the allowable percentage execution time increases. We observe that, energy savings achieved by VS3 can be increased by allowing an increase in execution time. For example, by allowing a 5% performance penalty, three out of four cases shown in Figure 10 gain about 5% more energy savings. This indicates that allowing some performance penalty can be an attractive option for energy-critical applications. We also observe that this trend slows down as the allowable performance penalty gets larger. This is due to the fact that

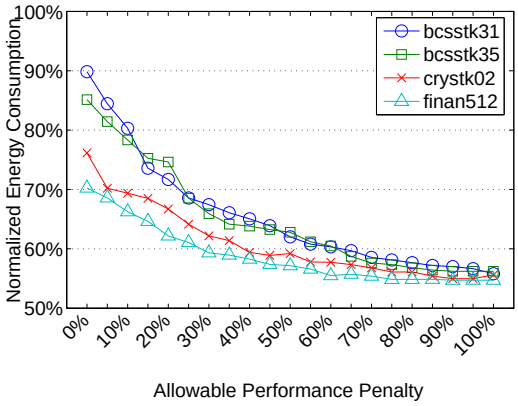


Figure 10: Normalized energy consumptions with VS3 when we allow some performance penalty. The x-axis represents the percentage execution time increase allowed. For clarity purpose, we present the results for only four cases. Similar trends are observed with other cases.

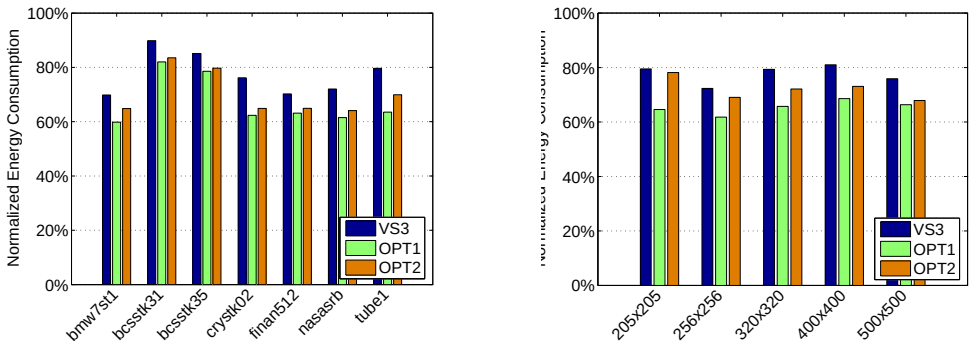
processor voltage has a lower limit due to technology, and we cannot scale a processor more if it has already reached the lowest possible voltage under which it can operate correctly. As allowable performance penalty increases, there are more and more cases where processors already reach the lowest voltage level. Consequently, the energy savings brought by voltage/frequency scaling increase more slowly. It can be observed that all the curves converge as the allowable performance penalty becomes very large, and this indicates that all processors reach their lowest voltage level in almost all execution stages.

4.2.3 Comparison with Optimum Results

It needs to be noted that, all our three schemes (VS1, VS2, and VS3) are heuristics in essence. Consequently, one might wonder how good our results are as compared with the optimum results. In this subsection, we explain and evaluate two optimum schemes, *OPT1* and *OPT2*, and compare them with our schemes. By such a comparison, one can learn how good the proposed schemes are in utilizing the inherent imbalance in the tree. Furthermore, the results of the optimum schemes set an upper bound for any voltage scaling scheme on such trees, and can indicate whether it is possible to employ better heuristics in this regard.

Optimum Scheme 1 — OPT1 is based on the assumption of perfect workload distribution and continuous voltage levels (both are unrealistic). Note that, in the original tree, the workload is not distributed evenly. Consequently, the overall execution time, T, is determined by the processor with the heaviest workload. Assume now that we could somehow re-distribute the total workload perfectly across the processors so that each processor has an equal amount of work (this might not be feasible in reality). Under this perfect distribution, the total execution time is reduced to T' ($T' < T$). Now, we can reduce the voltage level (and frequency) of the tree as a whole, and the total execution time of the tree can be increased. Since we also assume continuous voltage levels in this optimum scheme, we can always find a voltage level such that the execution time of the new tree reaches T, i.e., the execution time of the original tree.

Optimum Scheme 2 — OPT2 is also based on the assumption of continuous voltage levels. In the original tree, the workloads of different processors are different, and the real execution time of each processor is different. The execution time, T, of the processor with the heaviest amount of work determines the overall execution time. We can scale each processor individually, so that all processors' execution become T (again, this might not be feasible in reality due to synchronization issues). While at the first glance two optimum schemes that can generate different results (as will be discussed shortly) seem counter-intuitive, note that their



(a) Practical problems. (b) Model problems.

Figure 11: Comparison of VS3 with optimum results. OPT1 and OPT2 is explained in Section 4.2.3.

optimality are defined under different assumptions (i.e., perfect workload distribution versus synchronization-free execution).

Figure 11 compares VS3 with these two optimum schemes. Figure 11(a) presents the results for the practical problems, and Figure 11(b) presents the results for the model problems. It can be observed that OPT1 is better than OPT2 in all the cases, which is what one can expect. Knowing that OPT2 sets a tighter upper bound for our schemes, we now focus on the comparison of VS3 and OPT2. First, we notice that the VS3 scheme performs very good because the difference between VS3 and OPT2 is small. Specifically, on the average, VS3 achieves 7% less savings than OPT2. Furthermore, since the savings achieved by OPT2 may not be feasible in reality, VS3 should be even closer to a realistic optimum scaling scheme (if one could be found). It can also be observed from Figure 11 that there is a correlation between VS3 and OPT2. In general, when OPT2 has good results, VS3 also has good results. This means VS3 is able to catch the inherent opportunities in these cases, and follow the OPT2 scheme very closely.

5 Integrated CPU and Link Voltage Scaling Algorithm

So far, we discussed voltage/frequency scaling schemes that target only CPUs. We now present our integrated algorithm that scales the voltage/frequencies of both CPUs and communication links on top of the voltage scaling algorithm explained so far. We first describe the system model, and then explain our integrated voltage scaling algorithm based on this system model. We then present our experimental results obtained using our integrated voltage scaling scheme.

5.1 System Model

For illustrative purposes, we use a weighed tree computation similar to that given in Section 4.1, but annotated with slightly different notation in order to capture both CPU and communication link weights instead of using the VTE notation used in explaining our CPU voltage/frequency scaling algorithms. An example weighted tree is illustrated in Figure 12. We use N_i to represent the nodes in the tree. Each leaf node, which represents a local computation phase, is assigned to one processor, p_i, as shown in Figure 12. The nodes, which represent the distributed phase, are assigned to the processors that also operate to the leaf nodes of their subtree nodes. For example, N_1 is assigned to 4 processors, from p_0 to p_3, since N_1 makes use of all the processors in that subtree. The pair of numbers inside a given node in this figure represents the computation and communication weights associated with that node, e.g., N_1 node is assigned 50 unit of computation load and 25 unit of communication load. Similarly, N_3 is assigned 90 and 10 units of computation and communication loads, respectively. In the

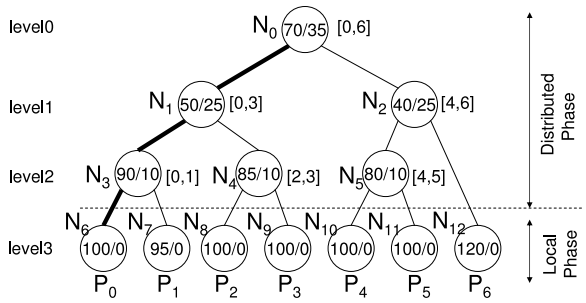


Figure 12: An example tree representing parallel sparse computations.

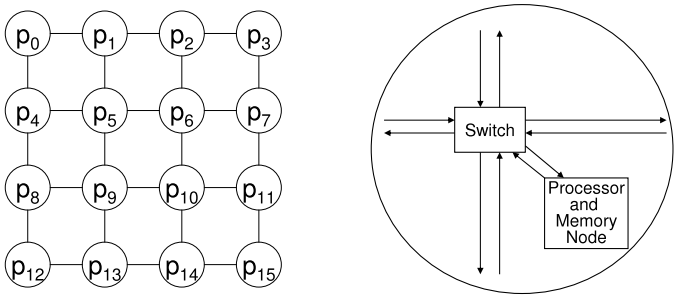


Figure 13: (a) Two-dimensional mesh topology (4 by 4). (b) A node with a bidirectional network.

tree-based computation model for sparse applications, the weight on each node is evenly distributed across all the processors assigned to that tree node. Based on this weighted tree notation, we can determine the *critical path*, which is represented as thick solid line in Figure 12. In other words, N_0, N_1, N_3 and N_6 constitute the critical path in this example tree, which determines the minimum execution time of the parallel application. The goal of this study is to reduce energy consumption of such tree-based matrix computations by taking advantage of the computation and communication exhibited by the tree nodes not in the critical path.

We focus on an $M \times N$ two-dimensional mesh based interconnection network as shown in Figure 13(a), though the analysis described in this work is applicable to other network topologies as well with proper modifications. Each pair of adjacent nodes in this 2D mesh topology is connected to each other using two unidirectional links. A node in this architecture typically consists of one or more processors, some amount of local memory, and a switch that routes messages through the nodes (Figure 13(b)). We use p_i to denote the id of the i^{th} node in this mesh network, which can be written as:

$$p_i = row(i) \times M + col(i), \tag{1}$$

where M is the row size of the mesh and $row(i)$ and $col(i)$ are the row and column positions, respectively, of the i^{th} node. For example, p_5 in Figure 13(a) can be represented using $(1, 2)$ since the size of this example mesh topology is 4×4.

Given the system architecture explained above, an application program considered in this study is parallelized using the message-passing interface, MPI [13]. In this model, the nodes communicate with each other using explicit send/receive commands. We propose integrated voltage/frequency scaling on CPUs and communication links. We apply our technique to these two components due to following reasons. First, both modern CPUs and communication links support voltage/frequency scaling circuits and we can make use of these capabilities to reduce power. Another reason is that each of these components are known to be a major contributor to total energy consumption in large parallel machines [23].

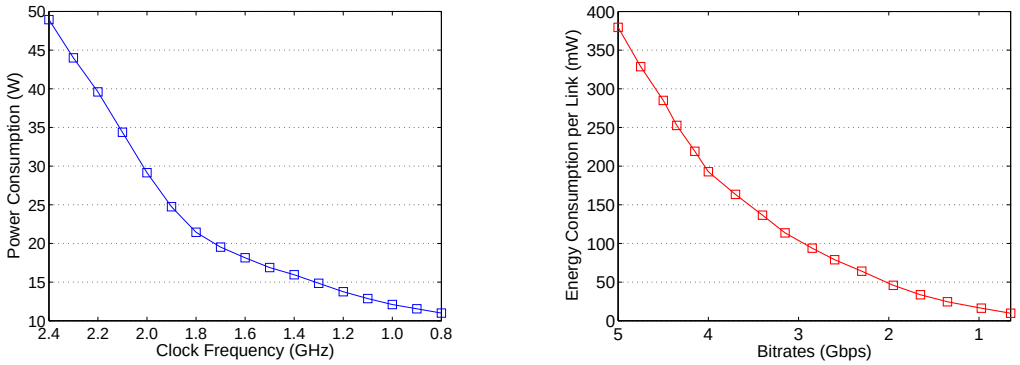


Figure 14: Left: CPU power model. Right: Link power model.

To illustrate the idea behind our approach that scales both CPU and communication link voltages in a coordinated fashion, let us first take a look at their energy/latency behavior. The dynamic power consumption, P, can be represented as:

$$P = 1/2 \cdot f \cdot N \cdot C \cdot V_{dd}^2, \tag{2}$$

where f is clock frequency, N is switching activity, C is effective capacitance, and V_{dd} is supply voltage. Therefore, the power consumption, P, is quadratically proportional to the supply voltage, V_{dd}, and frequency, f[1]. As shown by Equation 2, we can obtain quadratic drop in power consumption with a linear reduction in the clock frequency (f) and the supply voltage (V_{dd}).

The energy versus clock frequency[2] curves for CPU and communication links are drawn in Figure 14. The CPU curve is adapted from AMD's Athlon-64 processor datasheet [1], whose clock frequency can range from 800MHz to 2400MHz and the corresponding supply voltage can range from $1.1V$ to $1.5V$[3]. The communication link curve on the other hand is generated using data collected from [22], which has a supply voltage range of $0.9V$ to $2.5V$. The corresponding bit-rate range is from 650Mb/s to 5Gb/s. We can see from this figure that frequency-power curves are *convex*, which means that, as voltage and frequency are scaled down, the additional energy savings gained drop quadratically. Therefore, it should be beneficial from the energy perspective to scale voltages/frequencies of both CPU and link in a balanced (coordinated) manner, instead of scaling the voltage/frequency of one of them aggressively.

5.2 Integrated Approach

We now explain our algorithm for simultaneous voltage/frequency scaling for CPUs and communication links in tree-based parallel sparse computations using the example given in Figure 12, which is actually extracted from one of programs we tested in our experiments. The goal of our algorithm is to find an appropriate voltage levels and the corresponding frequency levels for CPUs and links that maximizes energy savings without impacting the overall execution time. Since our computation model is based on a tree, we can apply one of algorithms described in Section 4.1 where only CPU voltage is scaled down to save energy. Recall that, among our three CPU voltage scaling algorithms, i.e., VS1, VS2, and VS3 given in Section 4.1, VS2 and VS3 are better than VS1 while there is no clear winner between VS2 and VS3. Our algorithm starts with VS2, the root-first approach, as given in that, but it needs to be applied carefully due to the conflicts between the different voltage levels chosen for the communication links. To better explain this, let us consider the communication patterns exhibited by the example computation tree shown in Figure 12. Note that, since the leaf nodes in our tree-based sparse computation model perform only computation, the communication starts from the nodes just above the leaf nodes, i.e., level 2 in the tree. One notable characteristic of communicating nodes is that they

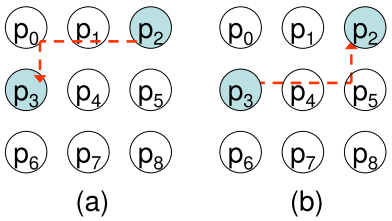


Figure 15: Mapping CG_4 to mesh topology.

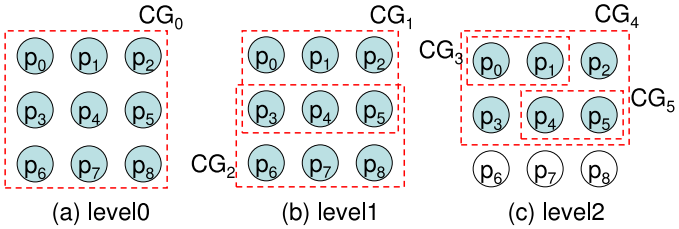


Figure 16: CGs mapped to the mesh topology.

can be grouped using a neighborhood concept. We use a CG (*Communication Group*) to represent the nodes that participate in communication at any point during execution, and a CG can be represented as follows:

$$[p_{low}, p_{high}], \tag{3}$$

where p_{low} is the processor whose id is the smallest among the processors in a given CG, and p_{high} is the processor whose id is the largest among the processors. Additionally, we use *size(CG)* to capture the number of processors in the CG. For example, in level 2 of the tree in Figure 12, there are three CGs (CG_3, CG_4, and CG_5), which can be represented as [0,1], [2,3], and [4,5], respectively. The size of all three CGs are 2 in this case. As we can see from this figure, the size of CGs becomes larger as we move to the lower levels, i.e., towards the root of tree. To see how these CGs are mapped onto the underlying mesh topology, let us consider the mapping between one of detected CGs, CG_4 in the level 2 of Figure 12, and the mesh topology, which is illustrated in Figure 15. Since we use an X-Y routing algorithm in communicating among nodes, each CG can be mapped to a set of rectangularly-connected nodes in the mesh topology. Therefore, we need to adjust p_{low} and p_{high} values of each CG using the actual processor-to-mesh topology mapping, and this can be represented as follows:

$$\begin{aligned} p'_{low} &= (min(row(\forall p_i)), min(col(\forall p_i))), \\ p'_{high} &= (max(row(\forall p_i)), max(col(\forall p_i))), \end{aligned} \tag{4}$$

where min() and max() functions give the minimum and maximum values of the column and row indices, respectively, of all the processors in each CG. Using this equation, we can obtain CG_4 mapped to the underlying mesh topology. Initially, CG_4 can be represented as [2,3]. As illustrated in Figure 15, CG_4 needs additional 4 processors (p_0, p_1, p_4, and p_5) to communicate each other using the X-Y routing algorithm. Since the row and column index for p_2 and p_3 are (0,2) and (1,0) respectively, we can redefine CG_4 as (0,0) and (1,2) using Equation (4) given above. All other detected CGs are marked using dashed rectangles in Figure 16 for each level of our example tree.

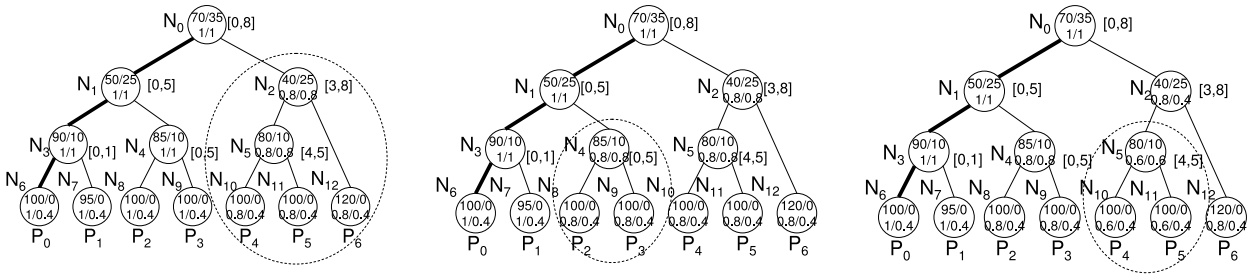
In the first step of our algorithm, we build a CG-annotated tree of the given parallel sparse matrix computation. Recall that, when we are given a tree representation of parallel sparse matrix computation such as the one in Figure 12, we already know the particular tree nodes that reside on the critical path of the tree. After obtaining all CGs, we then move to determine *conflict group*, denoted as D in this paper, to capture whether

```
VoltageScalingMain (node) {
   BuildCG (node);
   VoltageScaling (node);
}

BuildCG (node) {
   if (node.isLeaf) {
      node.plow = node.phigh = node.processor;
      CG = [node.plow, node.phigh];
   }
   else {
      node.plow = ( min (row(∀p_i ∈ CG)), min(col(∀ p_i ∈ CG)) );
      node.phigh = ( max (row(∀p_i ∈ CG)), max(col(∀ p_i ∈ CG)) );
      CG = [node.plow, node.phigh];
      build_CG (node.left);
      build_CG (node.right);
   }
}

VoltageScaling (node) {
   if (node.isLeaf) {
      // assign the lowest link power level
      node.linkLevel = MIN_LINK_LEVEL;     return;
   }
   else { // node is not leaf, i.e., node has children
      // determine critical node
      if (node.left.treeTime < node.right.treeTime ) {
         fastNode = node.left;     slowNode = node.right;
      }
      else {
         fastNode = node.right;     slowNode = node.left;
      }
      D = CG_i ∩ CG_x; // CG_x is the CG in critical path
      // select the slowest level for both CPU and link simultaneously.
      while ( (currCpuLevel < MIN_CPU_LEVEL) &&
            (currLinkLevel < MIN_LINK_LEVEL) &&
            (node.time < node.treeTime)) {
         currCpuLevel- -;
         ∀p_i ∉ D currLinkLevel- -;
      }
      // communication is dominant → reduce CPU voltage further
      if (fastNode.origTotalCommTime > fastNode.origTotalCompTime) {
         while ((currCpuLevel < MIN_CPU_LEVEL) &&
            (node.time < node.treeTime)) {
            currCpuLevel- -;
         }
      else { // computation is dominant → reduce link voltage further
         while ((currLinkLevel < MIN_LINK_LEVEL) &&
            (node.time < node.treeTime)) {
            currLinkLevel- -;
         }
      }
      node.cpuLevel = currCpuLevel;
      node.linkLevel = currLinkLevel;
      // recalculate total time based on newly determined voltage
      // levels and update node with the scaled voltage/frequency
      VoltageScaling (node.left); // perform VoltageScaling() on left node.
      VoltageScaling (node.right); // perform VoltageScaling() on right node.
   }
}
```

Figure 17: Integrated CPU/link voltage/frequency scaling algorithm.



(a) 1st phase. (b) 2nd phase. (c) 3rd phase.

Figure 18: Example application of our integrated CPU/link voltage scaling approach. The dashed circle is the subtree nodes being scaled in the corresponding phase.

there is any conflict among the CG groups that sit in the same level. We say that there is conflict between nodes CG_x and CG_i if the following condition holds true:

$$(D = CG_i \cap CG_x) \neq \emptyset \ \wedge \ \text{level}(CG_i) = \text{level}\ (CG_x), \tag{5}$$

where CG_x is a CG in the critical path. In Figure 16(b), CG_1 and CG_2 are in conflict because three processors, namely p_3, p_4, p_5, are shared by both CG_1 and CG_2. Note that the root node is always in the critical path so that all the nodes should operate under the maximum available frequency, i.e., maximum voltage level supported by the architecture (see Figure 16(a)) when they work on the root node. For level 1, we are able to reduce the voltage levels of processors that belong to CG_2, which is not in the critical path, and do not belong to the set of conflicting processors in CG_2. In our example, we can reduce the voltage levels of processors p_6, p_7, and p_8. It should be mentioned that the slack in a given node must be large enough to scale both CPU and link voltages. Once we assign determined voltage levels, we then recalculate the slack at each node in the tree. Our algorithm continues this way until all the nodes of the tree are processed. Note that, at the leaf nodes, we scale down all link voltage to the lowest levels because no communication is involved at leaf nodes.

The algorithm given in Figure 17 follows the approach explained above. Basically, our algorithm scales down a subtree from the root node, which is not in the critical path, as a whole. The algorithm starts by generating the CG-annotated tree by invoking the BuildCG function, followed by calling the VoltageScaling function. This function is invoked recursively, starting from the root node. If the node currently being processed is a leaf, our algorithm assigns the lowest voltage levels to all the communication links in the mesh. If the input node has a child node and one of its subtrees has slack, we scale down both CPU and link voltages at the same time until the point where the scaled execution time becomes very close to the original execution time. After scaling voltages simultaneously, if the remaining slack is large enough to scale down either CPU or link voltages, we further apply voltage scaling on that node. The decision on whether to scale down CPU voltage or link voltage is made based on their contribution to the total execution time of that node. More specifically, if the total computation time is longer than the total communication time, we scale down the link voltage because scaling down the component whose contribution is larger tends to consume the observed slack more quickly. So, it is better to scale the link voltage from the energy perspective, while utilizing the slack efficiently. On the other hand, in the case where communication is dominant time consumer for the node being processed, we scale down the CPU voltage. Our algorithm continues in this fashion until all the nodes of the tree are processed.

Figure 18 shows how our approach works in practice. For illustrative purposes, we use the normalized voltage and power numbers for both the CPU and link, which are given in Table 1. The initial voltage/power numbers are 1/1, as shown inside each node in Figure 18. In the first phase, we scale down the right subtree

Table 5: Default simulation parameters for the communication link.

Parameter	Value
Link frequency range	130MHz $\sim$ 1GHz
Number of voltage/frequency levels	5
Number of multiplexing stage	5
Bitrates per link	650Mb/s $\sim$ 5Gb/s
Link supply voltage range	0.9 $\sim$ 2.5V
Active link energy consumption	10.2 pJ/bit
Idle link energy consumption	8.5 pJ/cycle
Link frequency transition latency	10μs (100 link cycles)

because the left subtree is in the critical path. Therefore, we scale the both link and CPU voltage of all nodes in the right subtree to one level lower, 0.8 in this example. Note that, the link voltage of all leaf nodes are set to the lowest levels because this does not increase execution time. In the subsequent phase, our approach scales down the voltage/frequency of the subtree whose root is N_4 (Figure 18(b)). Lastly, the subtree rooted at N_5 can be scaled down further by using the slack present in that subtree (Figure 18(c)).

5.3 Experimental Evaluation

5.3.1 Experimental Setup

We use the same simulation platform described in Figure 8 except that the trace data being fed to the energy simulator indicate not only the computation involved at each level of tree, but also the communication load and patterns at each level of tree nodes. To obtain the energy consumption of network links, we use an energy model similar to that described in the literature [21, 22], and Table 5 gives the default simulation parameters for network links. While the circuitry associated with the network links (e.g., buffers, cross bar, etc) also consumes a certain amount of power, we do not account for this because router power consumption does not vary too much with and without the network links that support dynamic voltage scaling. This is because a flit remaining longer in a router due to slower links does not increase the energy consumption of the buffer read/write power nor the cross bar power [30]. Hence, when calculating the energy consumption of each program, we consider only the energy dissipated by CPU and network links. All other simulation parameters are fixed as given in Table 2.

We conduct experiments with same parallel sparse matrix solvers given in Table 3 and Table 4. Remember that the seven solvers described in Table 3 are from practical solvers and the five additional solvers in Table 4 are model solvers to study the sensitivity of our approach to the increased problem size. Table 6 presents the communication characteristics of these parallel sparse matrix solvers experimented in this section. The first seven rows of Table 6 correspond to the practical solvers given in Table 3. The remaining five rows correspond to the model solvers given in Table 4. The number of computing nodes (i.e., processors) and the size of mesh network used in each solver are given in the second and third columns of Table 6, respectively. The fourth and fifth columns of the table show the number of messages communicated among processors during computation and the total data volume of the communicated messages, respectively. Note that, the numbers given in these two columns are the average weight per processor. So, the total weight is dependent on the number of processors involved in each tree node. The last column is the contribution of the communication time to the total execution time. We can see from this table that the communication time (correspondingly communication volume and the number of messages) increases when more processors are involved in parallel execution. Since we use square mesh networks, our energy simulator takes into account the energy consumption of the CPUs and links that are actually used.

To evaluate the effectiveness of our approach, we conducted experiments with the following three schemes:

Table 6: Communication characteristics of parallel sparse solvers evaluated given in Table 3 and Table 4.

Solver Name	Number of Processors	Mesh Size	Number of Messages	Communication Volume (MB)	Percentage of Communication Time
bmw7st1	64	8×8	24,337	406.41	50.7%
bcsstk31	28	6×6	4,645	18.36	31.3%
bcsstk35	17	5×5	6,999	43.13	22.3%
crystk02	11	4×4	2,227	7.05	9.8%
finan512	28	6×6	7,364	39.59	44.7%
nasasrb	22	5×5	2,997	10.13	6.1%
tube1	7	3×3	2,557	12.16	8.7%
205x205	3	2×2	291	0.29	0.2%
256x256	7	3×3	648	0.82	3.8%
320x320	15	4×4	1,294	2.1	22.6%
400x400	31	6×6	2,318	4.5	38.1%
500x500	63	8×8	3,172	7.1	39.1%

- CPU-VS: This scheme scales down only CPU voltages, using the VS2 algorithm described in Section 4.1. It simply takes advantage of available computation slacks.
- LINK-VS: This scheme uses the same VS2 algorithm except that it is applied to scale down only link voltages based on the communication slacks available. The selection of link voltage level is made based on the algorithm explained in Section 5.
- CPU-LINK-VS: This scheme, which is the main contribution of this work, scales both CPU and link voltages using the algorithm given in Figure 17. If there is enough slack, this scheme tries to scale down both CPU and link simultaneously. When a voltage level chosen for one CG is not the same as those of the other CGs that share processors for communication, CPU-LINK-VS chooses the largest voltage level among the voltage levels of all the CGs, in an attempt to minimize potential performance overheads.

5.3.2 Results

Figure 19 gives the normalized energy savings with the three different schemes described in Section 5.3.1. All bars of a given solver are normalized with respect to the execution when *no* voltage/frequency scaling is applied. We can see from this figure that the energy savings obtained from both CPU-VS and LINK-VS are significant. Specifically, the average energy savings by CPU-VS and LINK-VS are 27% and 23%, respectively. This shows that scaling down either CPU or link voltage can be very effective in reducing total energy consumption. On the other hand, the CPU-LINK-VS scheme, which scales down CPU and link in a coordinated fashion, achieves 40% energy saving on average. This result clearly shows that it is better to scale voltages of both CPUs and links in an integrated manner rather than scaling only one of them aggressively, due to the diminishing energy saving rates, already demonstrated earlier by Figure 14. Note that, since all three schemes try to scale down the voltages/frequencies of the tree nodes that are not in the critical path, no schemes incurs any observable performance degradation.

In our next set of experiments, we perform a sensitivity analysis to see how the energy savings achieved by our approach are affected with the increase in the number of voltage/frequency levels supported by the underlying architectures, and the number of processors. To study the effectiveness of our approach with finer voltage levels in CPU and links, we experiment with 5 (our default value), 9, 17, and 33 voltage levels. The intermediate voltage levels are obtained by curve fitting based on the initial voltage/frequency points. All other simulation parameters are fixed as in Table 2 and Table 5. The normalized energy savings for all seven solvers used in our experiment under the different number of voltage levels are given in Figure 20. As one can observe

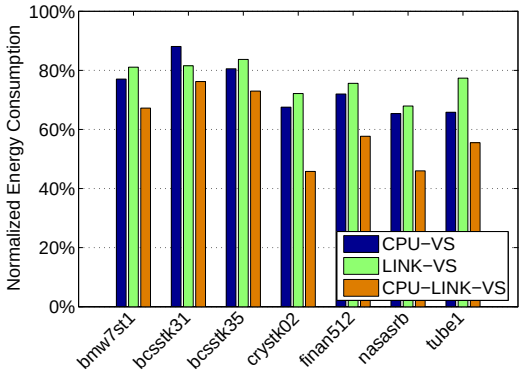


Figure 19: Normalized energy consumptions with the different schemes.

from these graphs, the energy savings obtained saturate as we increase the number of voltage/frequency levels. This is an anticipated result since finer granular voltage levels give more opportunity to scale down voltage levels, even when we have small slacks. However, we also see that energy savings start to saturate when the number of voltage levels reaches 17 or so. This shows that our scheme makes use of slacks in the tree successfully with reasonable number of voltage/frequency levels.

In the next set of experiments, we vary the number of processors and, correspondingly, the size of our two-dimensional mesh topology. We used the two set of model solvers in this experiment with five different processor sizes: 3, 7, 15, 31, and 63. In the first set of model solver, we try to keep the workload per processor constant as the number of processors increases. In the second set of model solver, on the other hand, we keep the total workload (when accumulated over all processors) constant as the number of processors increases. Figure 21(a) presents the results for the first set of model solver, whereas Figure 21(b) presents the results of the second set of model solver. Recall that we do not consider the energy consumption of the unused CPU nodes and the communication links connected to them, and the results presented in Figure 21 are the normalized energy consumption with various processor sizes. We can see from these figures that, as we increase the number of processors, the energy savings achieved by all three schemes decrease. The reason why the energy savings achieved by CPU-VS decrease is that, as the number of processors increases, overall execution time is dominated by communication, thereby decreasing the opportunities for scaling down the CPU voltages. Similarly, the energy savings achieved by LINK-VS also decrease due to the increased network contention brought by the larger number of processors, and network topology prevents the possibility to scale down the link voltages. Lastly, the energy savings obtained through the CPU-LINK-VS also decreases but this scheme gives the best energy savings for the all fives cases tested. It can be also observed that, in case of Figure 21(b) where we keep the workload assigned to each processor constant as the number of processor increases, the decrease in energy savings is saturated when the number of processor reaches 31. This is because the constant workload per processor tends to generate less communication overhead as the number of processor increases.

6 Conclusions

This paper makes two major contributions. First, it proposes several CPU voltage/frequency scaling schemes for parallel sparse computations. Second, it presents an algorithm that scales voltages/frequencies of CPUs and communication links in a mesh-based parallel system in a coordinated (integrated) fashion such that energy savings are maximized and performance is not affected. To test our algorithm, we implemented it and applied it to a set of tree-based sparse computations. The experimental results collected are very promising and show

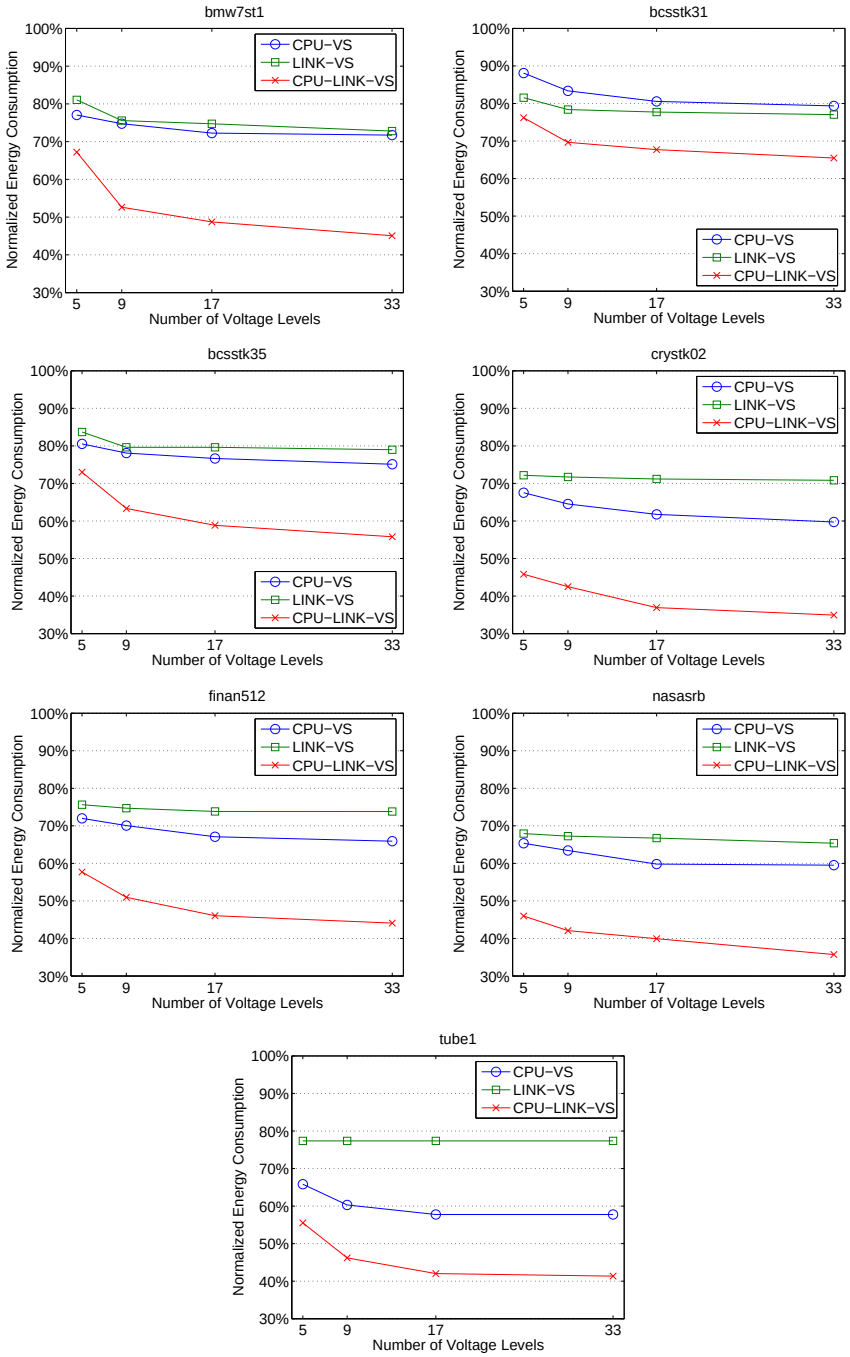


Figure 20: Normalized energy consumptions with the different schemes as the number of voltage/frequency levels vary.

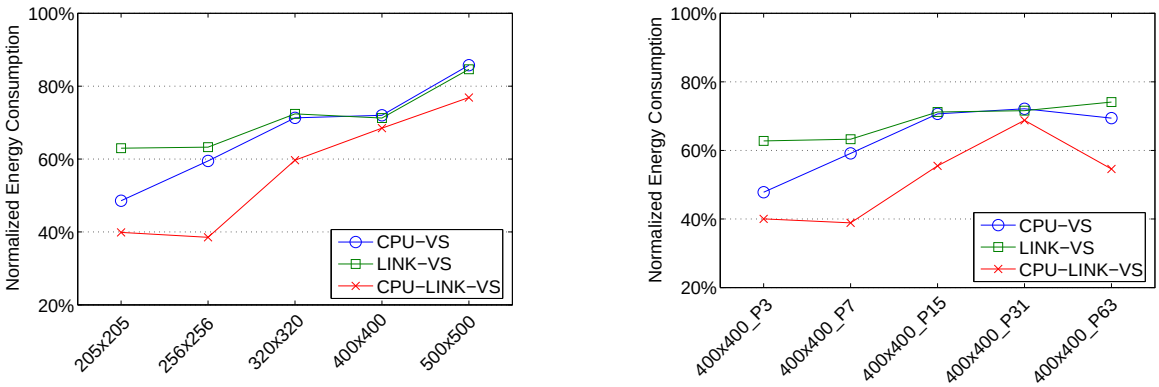


Figure 21: Normalized energy consumption with model problems as the number of processors increase. (a) Workload per processor is kept constant as the number of processors increases. (b) Total workload is kept constant as the number of processors increases.

that integrated CPU/communication link voltage scaling can generate much better results than the CPU voltage scaling alone and the link voltage scaling alone. Our results also show that the energy savings are consistent with the different problem sizes and different sets of voltage/frequency levels.

Acknowledgments

This work is supported in part by NSF grants CCF 0444158, CNS 0406340, CCF 0444345, and CCF 0102537.

Notes

[1]Since the clock frequency, f, can be represented in terms of V_{dd} and threshold voltage, V_t, as the frequency is reduced, the supply voltage can be reduced proportionally.

[2]In case of a serial link, the frequency dictates the bit-rates.

[3]Since the AMD datasheet states only TDP (Thermal Design Power), which is 89 W, we estimate the peak power consumption of the CPU for our study to be approximately 50 W based on our experience.

References

[1] Advanced Micro Devices, Inc. AMD Athlon 64 Processor Power and Thermal Data Sheet, 2004.

[2] A. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.

[3] J. Chase, D. Anderson, P. Thackar, A. Vahdat, and R. Boyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 103–116, October 2001.

[4] G. Chen, K. Malkowski, M. T. Kandemir, and P. Raghavan. Reducing Power with Performance Constraints for Parallel Sparse Applications. In *Proceedings of International Parallel and Distributed Processing Symposium*, April 2005.

[5] X. Chen and L. Peh. Leakage power modeling and optimization in interconnection networks. In *Proceedings of the International Symposium on Low Power and Electronics Design*, pages 90–95, August 2003.

[6] J. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report CSL–94–14, Xerox Palo Alto Research Center, 1995.

[7] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the Power-Hungry Disk. In *Proceedings of the USENIX Winter Conference*, pages 292–306, 1994.

[8] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient Server Clusters. In *Proceedings of the Second Workshop on Power Aware Computing Systems*, February 2002.

[9] M. Elnozahy, M. Kistler, and R. Rajamony. Energy Conservation Policies for Web Servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[10] V. W. Freeh and D. K. Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173, 2005.

[11] J. A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

[12] L. Grigori and X. S. Li. A new scheduling algorithm for parallel sparse lu factorization with static pivoting. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18. IEEE Computer Society Press, 2002.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[14] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. PSPASES: An Efficient and Scalable Parallel Sparse Direct Solver, 1999. `http://www-users.cs.umn.edu/~mjoshi/pspases`.

[15] A. Gupta, V. Kumar, and A. Sameh. Performance and Scalability of Preconditioned Conjugate Gradient Methods on the CM-5. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 664–674, 1993.

[16] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: Dynamic Speed Control for Power Management in Server Class Disks. In *Proceedings of the International Symposium on Computer Architecture*, pages 169–179, June 2003.

[17] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.

[18] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.

[19] Intel XScale (tm) Core Developer's Manual. `http://developer.intel.com/design/intelxscale/`, 2002.

[20] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1995.

[21] E. J. Kim, K. H. Yum, G. Link, C. R. Das, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Energy Optimization Techniques in Cluster Interconnects. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 459–464. ACM, August 2003.

[22] J. Kim and M. A. Horowitz. Adaptive Supply Serial Links with sub-1V Operation and Per-pin Clock Recovery. In *Proceedings of International Solid-State Circuits Conference*, February 2002.

[23] J. Luo, L.-S. Peh, and N. Jha. Simultaneous Dynamic Voltage Scaling of Processors and Communication Links in Real-time Distributed Embedded Systems. In *Proceedings of the Design Automation and Test in Europe Conference*, pages 1150–1151, 2003.

[24] K. Malkowski and P. Raghavan. Multi-pass Mapping Schemes for Parallel Sparse Matrix Computations. In *International Conference on Computational Science (1)*, pages 245–255, 2005.

[25] E. Ng and P. Raghavan. Towards A Scalable Hybrid Sparse Solver. *Concurrency: Practice and Experience*, 12:1–16, 2000.

[26] A. Pothen and C. Sun. A mapping algorithm for parallel sparse cholesky factorization. *SIAM J. Sci. Comput.*, 14(5):1253–1257, 1993.

[27] P. Raghavan. *Distributed sparse matrix factorization: QR and Cholesky factorizations*. PhD thesis, Pennsylvania State University, 1991.

[28] P. Raghavan, K. Teranishi, and E. Ng. A latency tolerant hybrid sparse solver using incomplete cholesky factorization. *Numerical Linear Algebra*, 10:541–560, 2003.

[29] Y. Saad. *Iterative Methods for Sparse Linears Systems*. PWS Publishing Co., Boston, MA, 1996.

[30] L. Shang, L.-S. Peh, and N. K. Jha. Dynamic Voltage Scaling with Links for Power Optimization of Interconnection Networks. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 91–102, 2003.

[31] D. Shin and J. Kim. Power-Aware Communication Optimization for Networks-On-Chips with Voltage Scalable Links. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 170–175, 2004.

[32] V. Soteriou and L.-S. Peh. Design-Space Exploration of Power-Aware On/Off Interconnection Networks. In *Proceedings of the IEEE International Conference on Computer Design*, pages 510–517, 2004.

[33] Transmeta. Crusoe Longrun Power Management White Paper. `http://www.transmeta.com/crusoe/longrun.html`.

[34] M. Weiser, A. Demers, B. Welch, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of Symposium on Operating System Design and Implementation*, pages 13–23, Nov. 1994.

[35] F. Worm, P. Ienne, P. Thiran, and G. D. Micheli. An Adaptive Low-Power Transmission Scheme for On-Chip Networks. In *Proceedings of the 15th international symposium on System Synthesis*, pages 92–100, 2002.