

Reducing Power with Performance Constraints for Parallel Sparse Applications *

G. Chen, K. Malkowski, M. Kandemir, and P. Raghavan
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA
{guilchen,malkowsk,kandemir,raghavan}@cse.psu.edu

Abstract

Sparse and irregular computations constitute a large fraction of applications in the data-intensive scientific domain. While every effort is made to balance the computational workload in such computations across parallel processors, achieving sustained near machine-peak performance with close-to-ideal load balanced computation-to-processor mapping is inherently difficult. As a result, most of the time, the loads assigned to parallel processors can exhibit significant variations. While there have been numerous past efforts that study this imbalance from the performance viewpoint, to our knowledge, no prior study has considered exploiting the imbalance for reducing power consumption during execution. Power consumption in large-scale clusters of workstations is becoming a critical issue as noted by several recent research papers from both industry and academia. Focusing on sparse matrix computations in which underlying parallel computations and data dependencies can be represented by trees, this paper proposes schemes that save power through voltage/frequency scaling. Our goal is to reduce overall energy consumption by scaling the voltages/frequencies of those processors that are not in the critical path; i.e., our approach is oriented towards saving power without incurring performance penalties.

1. Introduction

Scientific computing focuses on algorithms and software systems to enable computational modeling and simulation for knowledge discovery and design across science and engineering disciplines. Examples include mod-

eling fluid flow [11] and generating material microstructures for predicting macroscopic properties [21]. Typically, complex and realistic models lead to computations on very large matrices, which dominate total application time. Consequently, it is necessary to utilize parallel processing on clusters of workstations or multiprocessors to solve such applications and their underlying matrix problems. In the last decade, research in scientific computing has led to the development of scalable algorithms and software for such core matrix problems on parallel computers. The principal focus has been on optimizing schemes for improved performance leading to library software that can execute several fundamental *dense* matrix operations at near machine-peak speeds [5] with near ideal balance of loads among processors. But, many applications do not lead to such underlying dense matrix formulations.

In a large class of simulations, the models are based on time-dependent nonlinear partial differential equations in two or three spatial dimensions. These are often solved with implicit Newton-type methods or semi-implicit schemes leading to *sparse* or *irregular* matrix computations. Significant progress has been made in recent years to realize scalable algorithms and high-performance implementations for such sparse computations. In many of these applications, the sparse matrix that represents the underlying computation is modeled as a *graph*. While every effort is made to balance the workload across the parallel processors through suitable graph partitioning [18], achieving sustained near machine-peak performance with close-to-ideal load balanced mapping is inherently difficult. We consider such applications when the underlying parallel computation can be modeled as a *tree* with specific paths to the root representing computation at each processor. Such an abstract model shows irregularities as imbalances in the tree [18].

* This work was supported in part by NSF grants CCF 0444158, CNS 0406340, CNS 0097998, CCF 0444345, and CCF 0102437.

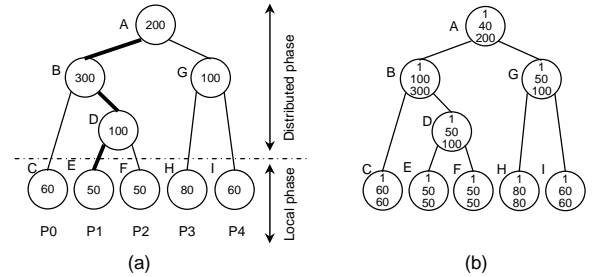
Recent trends show that power/energy consumption is becoming a critical issue for high-end large-scale parallel systems. We can attribute this to the following reasons. First, the power consumption of large server-class systems is reaching mega-watts levels, and thus contributes to a significant fraction of electric bills. Second, high power consumption requires sophisticated cooling mechanisms, which are typically too costly to purchase and even more costly to maintain. Third, allowing processors to operate on high power for long periods of time can hurt overall system reliability and availability. And, finally, from an environmental perspective, one may want to reduce power consumption as much as possible. Recent research proposed several strategies to reduce power consumption in high-performance large-scale systems (e.g., [8] and [26]). However, none of the prior efforts, to our knowledge, exclusively focus on sparse/irregular matrix computations, which exhibit very different characteristics from both dense scientific applications and web-based commercial applications.

In this paper, we propose a *voltage scaling* based energy reduction scheme for tree-based parallel sparse computations. Our 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 [10]. We note that many state-of-the-art processors (e.g., [6]) employ mechanisms that support voltage/frequency scaling; and in large parallel systems built from such components, each processor can be voltage/frequency scaled independently of the 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.

The remainder of this paper is structured as follows. Section 2 presents the tree-based computational model for sparse scientific computing problems. Section 3 presents our two voltage/frequency scaling algorithms. Section 4 discusses related work, and Section 5 concludes the paper with a summary of our observations.

2. Tree-Based Computational Model

Consider the simulation of a time-dependent phenomenon on a two or three dimensional spatial domain Ω modeled using partial difference equations. The domain Ω is typically discretized using finite-difference or finite-element schemes, and then a system correspond-



(a) Weighted tree. The numbers written inside the nodes indicate the associated computational costs. (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.

Figure 1. An example weighted tree and its VTE (Voltage-Time-Energy) tree.

ing to such a spatial discretization may be solved at each time-step until a steady state is achieved etc. In many applications, the spatial discretization employed can be highly irregular to allow finer resolution in regions of interest and coarser approximations elsewhere. Such irregularity is reflected in the associated sparse matrix, corresponding, for example, to linear system solution which may be required at each time-step. In this paper, we consider parallel sparse linear system solution as a model application to study the role of explicit power tuning strategies without impacting performance. Such computations typically dominate the execution time of many large-scale applications on multiprocessors and clusters of workstations.

There are many classes of parallel sparse linear solvers and it has been commonly observed that there is no single method that is consistently superior across application domains and computing platforms. Important classes include parallel direct solvers based on sparse factorization [7], iterative solvers [17], direct-iterative hybrids through preconditioning [12], domain-decomposition schemes, which are directly related to the solution of partial differential equations [19], and an emerging class of solvers that combine multiple methods [2]. However, despite the differences across classes and their constituent methods, a common element is that the sparse matrix can be represented as a graph, which can then be appropriately partitioned for parallel computation. The partitioning [18] 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 of the underlying parallelism and data dependencies.

We consider tree-based parallel sparse computations that are representative of parallel sparse solvers when the matrix is symmetric positive definite (solvers for non-symmetric systems include many extensions but are sim-

```

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. Algorithm I. 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.

ilar at a broad level). Parallel direct solvers are based on Cholesky factorization [1], and parallel direct-iterative hybrids use preconditioning with incomplete Cholesky factors [23]. Such solvers have a symbolic first phase followed by a numeric phase [14]. The symbolic phase comprises graph techniques to maintain sparsity, to partition the matrix for parallel computation, and to determine the actual structure of the Cholesky factor [15]. The numeric phase is of dominant cost, and it includes computing the sparse factor and using it to solve for a right-hand-side vector [15]. Further details of the symbolic phase are beyond the scope of this paper, but we describe the numeric phase to derive a suitable computational model.

The numeric phase involves computing either the sparse Cholesky factor L ($A = LL^T$) or its incomplete variant (\hat{L}), which contains only a limited subset of elements of L . The columns of L or \hat{L} can natu-

rally be grouped into *supernodes*. Each supernode contains a set of consecutive columns, with essentially the same zero/nonzero structure. More specifically, if a supernode contains columns $i, i + 1, \dots, j$, then the lower triangular submatrix of L induced by these columns and the corresponding rows is dense. For an incomplete form, \hat{L} has several smaller dense blocks in each supernode. These dense blocks are used to express sparse matrix-vector operations in terms of dense matrix operations; and thus allow the use of cache-efficient computational kernels as in the case of BLAS [20].

The overall numeric phase can be organized as parallel computations on *tree of supernodes*. Each tree node represents a supernode of L or \hat{L} and its corresponding set of dense-matrix operations. The tree structure dictates the data-dependencies; the computation at a node depends only on data associated with the subtree rooted at the node and its children supernodes, and the computations in disjoint subtrees can proceed independently and in parallel. Typically, the dense matrix computations associated with each node decreases from the root towards the leaves. Consequently, it is a common practice to exploit task-parallelism at lower levels of the tree and data-parallelism at nodes closer to the root. The root supernode is typically assigned to all processors. Next, each child supernode of the root is assigned to a subset of the processors, whose size proportional to the computations in the subtree. This process is repeated until each processor is assigned an independent subtree; such a subtree represents local-phase computations that proceed in parallel on each processor. At ancestor nodes of such local-phase ‘leaf nodes’, all processors assigned to the subtree rooted at the node participate in dense distributed matrix computations. Assignments of subtrees to processors are computed using the proportional mapping schemes [10] with weights on the tree to represent computation costs. Such schemes can be refined through several passes [22] to improve load-balance and to include costs/weights that model both communication and computation costs. However, despite such schemes, *inherent irregularities* in the sparse matrix will lead to imbalances.

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.

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

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

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 processor 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.

3. Voltage/Frequency Scaling 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) [4].

In this section, we present the algorithms for dynamically varying (scaling) CPU speed and voltage in order 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.

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 volt-

age/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 *Algorithm 1*, 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 a 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-

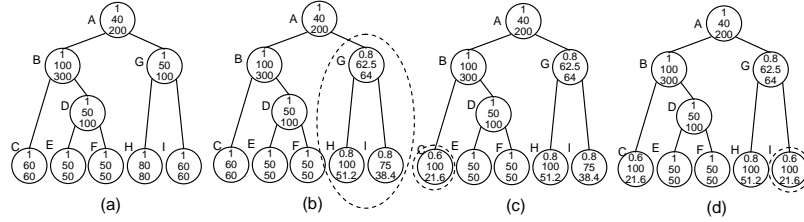


Figure 3. Example application of Algorithm I. (a) The original VTE tree. (b)-(d) Different steps of applying Algorithm I to the tree. The subtree or node in dashed circle is the one being scaled in the corresponding step.

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 Algorithm I 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 Algorithm I is $O(LN)$. Figure 3 illustrates how Algorithm I 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 Algorithm I, 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 Algorithm I 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, Algorithm I is no longer the optimum choice. For example, in Figure 3(d), we can scale node G down further as shown in Figure 4(a), or scale 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 Fig-

ure 5, we give *Algorithm II*, which exploits such opportunities. To obtain voltage assignments for a tree rooted at node A, we call VoltageScaling(A, 0). Note that, in Algorithm II, under a given allowable performance loss (which can be 0 meaning no performance loss), we first try to scale the whole tree as much as possible (as in the case of Algorithm I). When we reach the point where scaling the whole 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 6(a) gives an algorithm, called *VS2*, which is the root-first version of Algorithm II. Figure 6(b) gives an algorithm, called *VS3*, which is the children-first version of Algorithm II. The common code portion for both these versions, which tries to scale the whole tree/subtree together as much as possible, is given in Figure 5, and hence is omitted from Figure 6. 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 6(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 6(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 7(top) gives an example application of VS2 to the VTE tree shown in Figure 3(b). The scenarios shown in Figure 7(top – a) and Figure 7(top – b)

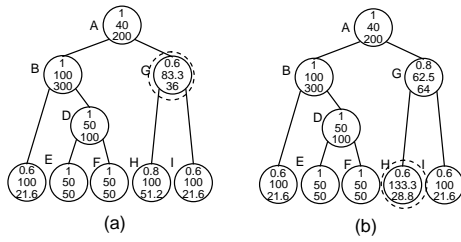


Figure 4. Two examples showing that the result achieved by Algorithm I in Figure 3(d) is not optimum. The nodes in dashed circles have been scaled further.

```

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.
  }
}

```

Figure 5. Algorithm II. The parameter *slack* is the allowable execution time increase (for the whole subtree being considered) after voltage/frequency scaling. The helper functions, including *TreeTimeAtLevel()* and *AdjustTreeVoltage()*, are defined in Figure 2.

are similar to those with Algorithm I, 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 Algorithm I, 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 Algorithm I since it saves more energy in computing node G. Figure 7 (bottom) gives an example application of VS3 to the VTE tree shown in Figure 3(b). The first two steps in Figure 7 (bottom) are the same as those in Figure 7 (top). 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(bottom – c)), we exploit the slack by scaling the two children first. After scaling the children, we cannot scale node G any further since there is not enough slack left for node G.

```

VoltageScaling(node, slack) {
  ...// The common code portion of Algorithm II
  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. Complexity: $O(LN)$, L : the number of voltage levels, N : the number of nodes in the tree.

```

VoltageScaling(node, slack) {
  ...// The common code portion of Algorithm II
  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) 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. Complexity: $O(LN)$, L : the number of voltage levels, N : the number of nodes in the tree.

Figure 6. Two versions of voltage/frequency scaling Algorithm II. The helper functions, including *AdjustTreeVoltage()*, *AdjustNodeVoltage()*, and *NodeTimeAtLevel()*, are defined in Figure 2. The common code portion for both versions is given in Figure 5.

In the *VoltageScaling()* functions of Algorithm II, the parameter *slack* indicates the maximum execution time increase (i.e., performance degradation/loss) allowed to save energy. Consequently, Algorithm II can work under a given performance degradation bound. Assuming that

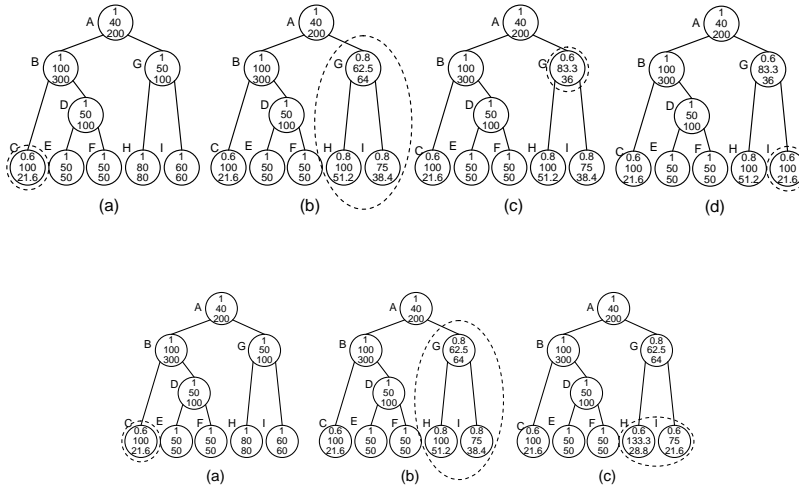


Figure 7. Example application of the children-first version of Algorithm II (VS2) and (VS3). Top: VS2, Bottom: VS3. The original VTE tree is given in Figure 1(b). (a)-(d): Different steps. The subtree or node in dashed circle is the one being scaled in the corresponding step.

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 $\text{VoltageScaling}(R, T * P)$ to obtain voltage assignments under such a performance constraint. Note that, when we use $\text{VoltageScaling}()$ with parameter 0, this indicates that we are not tolerating any increase in the original execution time. Algorithm I can also be implemented in a slack-based fashion, and thus can also be used in cases where performance constraints are specified. The slack-based version of Algorithm I is not presented here explicitly due to space concerns.

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.

4. Related Work

Power optimization for high-performance systems is an active research topic. Pinheiro et al [25] propose dynamic reconfiguration techniques for cluster-based servers, in which cluster nodes are turned on or off to adjust to the dynamic changes in system load. Elnozahy et al [8] use dynamic voltage scaling and request batching to reduce the energy consumption of web servers. Flautner et al [9] propose a voltage scaling policy that sets CPU speed on a per-task basis. These techniques are proposed for various kinds of server loads, and are not suitable for parallel sparse computations. Our work, on the other hand, is the first to address the energy optimiza-

tion problem for tree-based sparse computation models. The energy consumption of disk arrays in servers have received a lot of attention. Gurumurthi et al [13] show that multi-speed disks can provide significant energy savings. Heath [16] et al use compiler-based application transformations to increase device idle times and evaluate the potential energy and performance benefits brought by such transformations. Carrera et al [3] study several power optimization approaches under a two-speed disk environment using real software and hardware. Pinheiro and Bianchini [24] present a technique that dynamically migrates popular disk data to a small number of disks to increase the opportunity of putting other disks into low-power modes. Research in these areas are orthogonal to our work, and how to incorporate them into our framework is in our future agenda.

5. Concluding Remarks

While sparse computations have received a lot of attention from the performance angle, their energy consumption trends have not been the focus of the past research. This is unfortunate since (1) a large fraction of scientific codes are built from inherently sparse computations, and (2) power consumption of large-scale applications executing on parallel multiprocessors and networks of workstations is becoming increasingly critical as these servers employ fast processors. Motivated by these observations, in this paper, we proposed and evaluated two algorithms that reduce power consumption of tree-based parallel sparse computations through voltage/frequency scaling.

References

- [1] C. Ashcraft et al. SParse Object Oriented Linear Equations Solver, 1999. <http://netlib.ccp14.ac.uk/linalg/spooles/spooles.2.2.html>.
- [2] S. Bhowmick, L. McInnes, B. Norris, and P. Raghavan. Robust algorithms and software for parallel PDE-based simulations. In *Proceedings of HPC 2004, The Twelfth Special Symposim on High Performance Computing at the 2004 Advanced Simulation Technologies Conference*. Arlington, VA, April 18-22, 2004. To appear.
- [3] E. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th International Conference on Supercomputing*, June 2003.
- [4] A. Chandrakasan and R. Brodersen. *Low Power Digital CMOS Design*. Kluwer Academic Publishers, 1995.
- [5] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. LAPACK Working Note 80. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines. Technical Report CS-94-246, Department of Computer Science, University of Tennessee, September 1994.
- [6] Crusoe Longrun Power Management White Paper. <http://www.transmeta.com/crusoe/longrun.html>.
- [7] 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.
- [8] 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.
- [9] K. Flautner and T. Mudge. Vertigo: automatic performance-setting for linux. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, December 2002.
- [10] 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.
- [11] W. Gropp, D. Keyes, L. McInnes, and M. Tidriri. Globalized Newton-Krylov-Schwarz algorithms and software for parallel implicit CFD. to appear in *Int. J. High Perform. Comput. Appl.*
- [12] A. Gupta, V. Kumar, and A. Sameh. Performance and scalability of preconditioned conjugate gradient methods on the CM-5. In R. F. Sincovec, D. E. Keyes, M. R. Leuze, L. R. Petzold, and D. A. Reed, editors, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 664–674, Philadelphia, PA, 1993. SIAM Publications.
- [13] 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.
- [14] M. T. Heath, E. Ng, and B. W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33:420–460, 1991.
- [15] M. T. Heath and P. Raghavan. Performance of a fully parallel sparse solver. *Int. J. Supercomputing Appl.*, 11:49–64, 1997.
- [16] T. Heath, E. Pinheiro, , and R. Bianchini. Application-supported device management for energy and performance. In *Proceedings of the Workshop on Power-Aware Computer Systems*, February 2002.
- [17] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952.
- [18] 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.
- [19] D. E. Keyes. Trends in algorithms for nonuniform applications on hierarchical distributed architectures. In M. D. Salas and W. K. Anderson, editors, *Proceedings of the Workshop on Computational Aerosciences for the 21st Century*, pages 103–137. Kluwer, 2000.
- [20] C. L. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.
- [21] Z.-K. Liu, L.-Q. Chen, Q. Du, S. Langer, P. Raghavan, J. Sofo, and C. Wolverton. Computational tools for multicomponent materials design, 2002–2007. NSF ITR project; <http://matcase.psu.edu/index.html>.
- [22] K. Malkowski and P. Raghavan. Data mapping techniques for parallel sparse factorization, 2004. Presentation at SIAM Conference on Parallel Processing for Scientific Computing 2004, San Francisco, California.
- [23] E. Ng and P. Raghavan. Towards a scalable hybrid sparse solver. *Concurrency: Practice and Experience*, 12:1–16, 2000.
- [24] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 68–78. ACM Press, 2004.
- [25] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Dynamic cluster reconfiguration for power and performance. *Compilers and operating systems for low power*, pages 75–93, 2003.
- [26] V. Soteriou and L.-S. Peh. Dynamic power management for power optimization of interconnection networks using on/off links. In *Proceedings of the 11th Symposium on High Performance Interconnects (Hot Interconnects)*, Stanford, CA, August 2003.