Algorithms Minimizing Peak Power on Mesh-Connected Systems

Quentin F. Stout

Computer Science and Engineering
University of Michigan

Correspondence:

Quentin F. Stout
Computer Science and Engineering
University of Michigan
2260 Hayward
Ann Arbor, MI 48109-2121
qstout@umich.edu

734/763-1518   734/763-8094 (fax)

Abstract

There are many situations in parallel computing where reducing power consumption is an important goal. We consider mesh-connected systems where each processor is connected to its neighbors in a regular grid, such as occurs in cellular automata, sensor networks, and some supercomputers, focusing on systems with myriad simple processors on a chip. Most algorithms for such computers assume that all processors are active at all times, an assumption which is often unrealistic when power is supplied externally. This leads one to develop algorithms minimizing the peak power. Optimal or near-optimal algorithms are developed for basic problems involving images, such as labeling the components and determining the distances between them, and graphs, such as determining a minimal spanning forest and deciding if a graph is biconnected. These algorithms also minimize the total energy, and can be modified to simultaneously reduce the total power used by any processor, a consideration of considerable importance in sensor networks.

Keywords: mesh-connected computer, low power, energy, parallel computing, image processing, graph algorithm, connected component, minimal spanning tree

1 Introduction

Systems connected in a grid fashion have occurred throughout parallel computing history, from cellular automata to sensor networks (typically an irregular grid) to interconnection networks for high-performance multicore chips [20] and supercomputers [3, 4, 10]. For such systems physical location and distance play an important role, as opposed to PRAMs or serial computers. More recently another physical consideration, power consumption, has taken on importance. It is a concern in systems such as cell phones, laptops, and supercomputers. The last is different from the others in that the power is external and does not diminish over time, but supplying the peak power required is a major concern. Removing the heat generated introduces packing constraints which in turn affect communication time, and hence various tradeoffs need to be made. For example, peak power considerations resulted in the BlueGene/L utilizing slow, but numerous, processors [4].

This paper addresses the problem of minimizing the peak power required for parallel algorithms for systems such as single chips with many tiny, simple processors. For example, there are image processing chips which both detect images and do substantial processing on them. It is not realistic to assume that the entire chip can have most of its circuits active at the same time. (E.g., see [7] for an image processing chip with processing capabilities and emphasis on energy efficiency.)

We utilize the basic *mesh-connected computer* model: the system has n processors arranged as an $\sqrt{n} \times \sqrt{n}$ grid, where each processor can communicate only with its immediate neighbors (either the 4 neighbors sharing an edge, or the 8 sharing an edge or corner). To simplify exposition we assume that n is a power of 4, with modifications to the more general case being straightforward. Each processor can store a fixed number of words of logarithmic length, and all operations on these words, including sending one to a neighbor, take constant time and energy. Each processor starts with its coordinates (x, y), $x, y \in [0, 1, \ldots \sqrt{n}-1]$. For purposes of timing analysis the system is SIMD, though strict synchronization can be relaxed. Note that algorithms for such a basic system can be simulated on a variety of systems.

We say that a processor is *using power* if it is calculating or communicating, and otherwise is not using power. While we say it is not using power it might actually be using a greatly reduced amount in a wait state, either counting to know when to awaken or to be woken by the arrival of a message from a neighbor. It will be seen that all of the algorithms have the property that a processor is only calculating in a fixed period around a message transmission, and hence to determine power utilization it will suffice to merely count messages.

Standard mesh-connected computer algorithms assume all processors are on all the time, and hence for a mesh of n processors peak energy is $\Theta(n)$ and total energy is the product of time and n. Many basic algorithms take $\Theta(\sqrt{n})$ time [1, 2, 9, 12, 13, 19] and $\Theta(n^{3/2})$ total energy. Note that this time is a lower bound for any nontrivial problem since it is the diameter of the communication network. As for total energy, $\Omega(n^{3/2})$ is a lower bound for operations such as sorting or matrix multiplication where $\Theta(n)$ values may need to be transported distance $\Theta(\sqrt{n})$. These operations can in fact can be completed in $\Theta(\sqrt{n})$ time using this much energy on a standard mesh-connected computer [19], and thus, for them, reducing peak power necessarily increases time. For many other problems, however, it will be shown that far fewer messages can be used while still finishing in $\Theta(\sqrt{n})$ time. When the total data movement is $o(n^{3/2})$ then there is the possibility of a solution in this time bound using only $o(n)$ peak energy.

Rather than directly indicating which processors are on at any given time, it is useful to think of trained *squirrels* traversing the mesh, where the presence of a squirrel indicates that the processor is on. The squirrels have a memory of a finite number of words, they can keep track of their location, and they can leave a finite number of words at any location. Admittedly squirrels are an unusual computing model, but since many of the algorithms require taking something from one place and leaving it at another, and then being able to go back and get it if needed, squirrels seem to have the requisite skills, though their ability to be trained to cooperate with one another remains an open question. A squirrel carrying information from one location to another corresponds to a sequence of communication steps, where both the number of steps (time) and total energy are proportional to the distance traveled, and the peak power is proportional to the maximum number of squirrels active at any one time. Squirrel algorithms have some similarities with pebble algorithms for automata [5, 6, 14]. Pebbles are used to keep track of positional information, and can be used to help traverse mazes and more general graphs. Here the problem descriptions, such as labeling components, require the ability to store words of logarithmic size, rather than the fixed size inherent in pebble algorithms, and hence the cellular automata model of pebble algorithms is not quite suitable.

Peak energy is merely the number of squirrels, denoted s. For any nontrivial problem for which every position must be visited at some point the total energy must be $\Omega(n)$, and hence if time = $\Theta(n/s)$ then the speedup is optimal, which implies that the peak energy vs. time tradeoff is optimal. Further, if this holds for $s = \Theta(\sqrt{n}\ )$ and information must be passed from some processor to another one distance $\Theta(\sqrt{n}\ )$ away then the time is the minimum possible no matter how much energy is expended. While thinking in terms of squirrels moving around simplifies the descriptions of many algorithms, in some cases adjacent squirrels may stay where they are and merely exchange information, and occasionally algorithms can result in two squirrels needing to occupy the same location at the same time. Note that in a valid algorithm for a mesh-connected computer, the number of squirrels per location cannot grow unboundedly with n. We also assume that a squirrel can determine which adjacent locations are occupied by squirrels. In most cases the algorithm obviously guarantees that the time for a squirrel to move from one location to another is equal to the distance between them, but in some cases multiple squirrels may have paths that depend on the data, and the paths end up overlapping. Standard routing algorithms can be used to ensure that the squirrels are still able to complete their tasks in the time claimed.

It is trivially true that any algorithm taking t time on an $\sqrt{m} \times \sqrt{m}$ mesh can be stepwise simulated in $\Theta(tm/s)$ time and total energy $\Theta(tm)$ by $1 \leq s \leq m$ squirrels, where each squirrel is responsible for stepwise simulation of a $\sqrt{m/s} \times \sqrt{m/s}$ submesh. This fact will be used in some of the algorithms in which a small subproblem is solved via a standard (power-oblivious) mesh algorithm.

The algorithms are described for $s = \sqrt{n}\ $, and it is easy to see how to simulate them with fewer squirrels with only linear slow-down. In general such simulation requires some care since in addition to the operations being performed the simulating squirrel must move from the location of one squirrel being simulated to the location of another, i.e., extra time is added. Contrast this with the result mentioned in the previous paragraph, which corresponds to starting with an algorithm of m squirrels which never move from their initial position, and then simulating them by s squirrels which can just do simple scans of their subsquare to move from simulating one of the original squirrels to the next. For the results in this paper, however, the simulation tends to be simple and details will be omitted. Further, in the general case one might need the simulating squirrel to have memory proportional the number of squirrels being simulated, or spend extra time storing information at the sites, but again that is not true here and the memory can be held constant no matter what the value of s is in relation to n.

A single squirrel cannot in general use an algorithm for a standard serial computer without increasing the time required since it is constrained by the physical dispersion of information, something normally ignored in serial algorithms (unless cache behavior or paging is a concern). Squirrels also introduce a constraint in that the pattern of activation in the parallel computer follows connected paths and does not jump around. While we don't make this assumption for the underlying power-constrained parallel computer model, we don't know of any problems for which this extra capability provides faster (in O-notation) algorithms, given that a processor can only decide to turn on based on the information it has, not information only known far away. There might be a way to exploit the fact that it didn't receive information, but that does not seem to be a widely useful capability.

Throughout we assume the squirrels have a unique integer id $\in \{0, \ldots, s-1\}$ and that each squirrel knows s and n. At each processor its (x,y) coordinates are stored. Often the processor's z-order index will be used. In z-ordering, the coordinates $(x_k x_{k-1} \ldots x_0,\, y_k y_{k-1} \ldots y_0)$ yield the index $y_k x_k y_{k-1} x_{k-1} \ldots y_0 x_0$. Hilbert curve ordering would be equally useful.

2 Image Algorithms

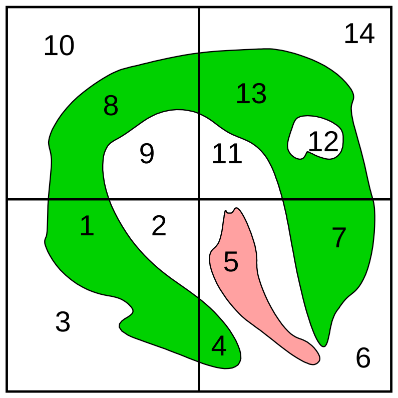
For image data we assume that a $\sqrt{n} \times \sqrt{n}$ image is stored one pixel per processor, where each pixel has a color. By a *figure* we mean a connected component of pixels of the same color, where we typically think of figures representing objects on a white background. We consider two pixels to be adjacent if and only if they share an edge, but this can trivially be expanded to include corner adjacency. By *labeling figures* we mean that each pixel is assigned a label, and that two pixels have the same label if and only if they are in the same figure. Initially each pixel starts with its label being its processor's z-index. The final label of the figure will be the minimal initial label of any of its pixels, and this position will be called the figure's *leader*.

A single squirrel can label the figures in $\Theta(n)$ time. To do so, it uses a simple z-ordered scan to traverse the image. When it encounters a pixel with label equal to its initial label then it has encountered a new figure. It can then use a depth-first search to label the figure in time proportional to its size. Once the figure is labeled the squirrel returns to figure's leader and resumes the scan. The scan takes $\Theta(n)$ time, and since the time to label all pixels in a figure is proportional to their number, the total labeling time is $\Theta(n)$, giving $\Theta(n)$ time for the algorithm.

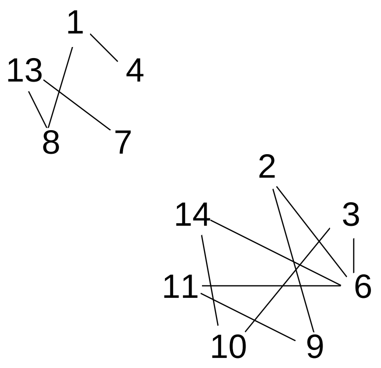
For multiple squirrels, however, we use a significantly different approach. A single figure may have $\Theta(n)$ pixels, and hence having only one squirrel work on it would not improve upon the time of a single squirrel labeling the entire image. Instead, we use a well-known divide-and-conquer approach (see [12, p.30] for a generic version of this approach). The algorithm is as follows: to solve the problem in a square, suppose the problem has been solved for the 4 subsquares. Within the larger square, the only figures where the pixels' labels are inconsistent are those that cross the borders between subsquares. The edges connecting the two sides, the ones that contain the information needed to make the labels consistent, form the edges of a graph in which vertices are the labels of pieces adjacent to the sides. In this graph the connected components need to be labeled (see Figure 1). To label this graph, move all of the edge information to the center and use an edge-based algorithm to label the components, and then move the information back to the edges. Ultimately the highest level is reached, and then the final labels are propagated by reversing the process.

When squares of size m are being worked on the movement of the subsquares' edge information to a submesh of size $m/4$ involves $\Theta(\sqrt{m})$ edges being moved a distance of $\Theta(\sqrt{m})$, for a total energy of $\Theta(m)$ per square. The edge-based component labeling involves a submesh of size $\Theta(\sqrt{m})$, and can be done in $\Theta(m^{1/4})$ time [17] and $\Theta(m^{3/4})$ total energy, and thus the movement to the center dominates the time and total energy. Since the movement to the center takes energy proportional to the area, and the squirrels are evenly distributed among the squares, each level of recursion takes the same time, $\Theta(n/s)$. There are $\log_4(n) - \log_4(n/s) = \log_4(s)$ levels of parallel recursion, so we arrive at the following for the energy range $1 \leq s \leq \sqrt{n}$:

Theorem 1 *Labeling the figures of an image of size n can be performed in $\Theta((n \log s)/s)$ time using peak power s, for $1 \leq s \leq \sqrt{n}\, \log n$.*



image



label graph

Figure 1: Merging subsquares to label figures

Proof: To finish the theorem we need to consider the energy range $\sqrt{n} \leq s \leq \sqrt{n} \log n$. Partition the mesh into subsquares of $r = (n/s)^2$ pixels. There are n/r such squares, so each can be assigned $sr/n = n/s$ squirrels. Note that this is $\sqrt{r}$, and hence by the above the subsquares can be labeled in $\Theta(\sqrt{r} \log r)$ time. Now a single merge step is used, merging all n/r squares at once. The number of squirrels is linear in the size of all of the boundaries of the squares, so the label information of all of the boundaries can be simultaneously moved to the center and made consistent, in $\Theta(\sqrt{n})$ time. As long as $\sqrt{r} \log r = \Omega(\sqrt{n})$, the time for labeling the squares dominates the total time. Since $\sqrt{r} \log r = 2(n/s) \log(n/s)$, the time is as claimed for $s \leq \sqrt{n} \log n$ □

This divide-and-conquer approach has recently been utilized for sensor network algorithms [16]. In Section 4 it will be shown that the above algorithm can be adjusted to achieve the minimum total power goals in [16] while retaining its peak power properties.

Since rodents are being used to perform the algorithms, the following seems appropriate:

Corollary 2 *Given a black/white maze of size n with start and stop sites, in $\Theta((n \log s)/s)$ time s squirrels, $1 \leq s \leq \sqrt{n} \log n$, can decide if the maze has a path from start to stop.* □

Note that this does not say that they determine the shortest path, merely that they can determine if there is a path. The power/time tradeoff for shortest paths is an open question, even for a single squirrel.

The algorithm in Theorem 1 is within a logarithmic factor of work-optimal parallelization, and it is an open question whether this factor can be eliminated. Further, when $s = \sqrt{n}$ the time is also slower than the optimal time by a logarithmic factor [13]. Once the figures have been labeled various properties of them can be determined without the extra logarithmic factor, but a bit more care is needed. The *large* figures, those having more than $\sqrt{n}$ pixels, are partitioned into pieces and the results on the pieces are combined to get the final result. The pieces are of size $\sqrt{n}$, and a squirrel will work on a piece and take the result directly to a location where the piecewise results are combined all at the same time, rather than combining them in the tree-like fashion used for labeling. A figure with $\sqrt{n}$ or fewer pixels will be called *small*.

Within a figure, the *rank* of a pixel is its position in the z-order numbering of the pixels in the figure (with the numbering starting at 0, i.e., the label's leader is rank 0). See Figure 2. A processor in a large figure with rank a multiple of $\sqrt{n}$ is a *breakpoint*. We say that an image is *strongly labeled* if in addition to being labeled, every processor contains the processor's rank in its figure and the number of pixels in the

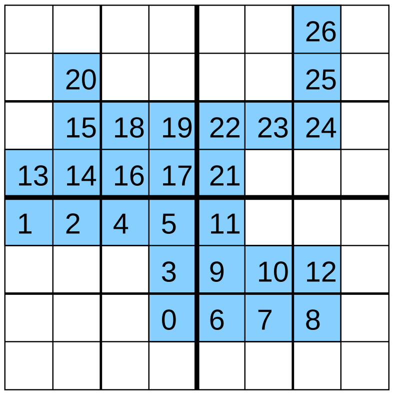


Figure 2: Rank ordering within a figure

figure. Each breakpoint also contains the location of the next breakpoint. It is straightforward to determine ranks as the figures are being labeled, without increasing the time, using the property that when squares are being merged they contain consecutive positions in the space-filling curve ordering, and hence merely knowing the number of processors in the subsquares allows one to determine the starting rank of processors in each subsquare.

For the processors that are breakpoints, once the ranks are known a simple bottom-up then top-down pass can be used to determine the location of the next breakpoint. A somewhat more complicated approach can reduce the time to $\Theta(\sqrt{n}\,)$. It introduces a technique that will be employed in more complex operations. Note that all of the large figures combined have at most $2\sqrt{n}$ breakpoints since each corresponds to a collection of $\sqrt{n}$ pixels, except for the last breakpoint in a figure which may be in a piece containing only itself. Thus if $s = \sqrt{n}$ we can assign each squirrel at most 2 breakpoints to be responsible for. However, a given region may have many breakpoints. To assign breakpoints to squirrels, first have the squirrels, one per row, move from right to left, counting the number of breakpoints encountered. Once these totals have been deposited in the leftmost column, it's simple to have each squirrel proceed bottom-up to determine which row(s) contains its breakpoints, and then within the row(s) determine which breakpoints it is assigned to. Temporarily, for the purposes of locating the next breakpoint, the pixel of highest rank within its figure is also treated as a breakpoint, and hence there may be nearly $3\sqrt{n}$ points involved, so each squirrel really carries 3 points.

Once each squirrel has located a breakpoint, it creates a record containing the breakpoint's label, rank, and location, and then carries this to a central $n^{1/4} \times n^{1/4}$ subsquare. A simple mesh algorithm is used to sort the records by label, and within each label by rank. If the sort is into alternating row major order (or any other contiguous ordering) then for each breakpoint the record of the next breakpoint in its figure is in an adjacent record. This information is added to the breakpoint's record, and then the squirrels carry the records back to the breakpoints and deposit the location of the next breakpoint, completing the operation.

By *broadcast over figures* we mean that there is a value at the leader which is then copied to every pixel in the figure. By *reduction over figures* we mean that there is a commutative semigroup operation $*$ over a set S, and that each pixel p has a value $v(p) \in S$. At the end of the reduction operation, the leader of figure F has the value $*\{v(p) : p \in F\}$. We assume that $*$ can be computed in unit time. Broadcast and reduction can easily be performed using the divide-and-conquer approach in Theorem 1, taking the same time bounds. Here, however, we remove the extra logarithmic factor.

Theorem 3 *Given a strongly labeled image of size n, broadcast and reduction over figures can be performed in $\Theta(n/s)$ time using peak power s, for $1 \leq s \leq \sqrt{n}$.*

Proof: Note that by using depth-first search a single squirrel can do broadcast and reduction over a figure in time proportional to the size of the figure. This will be used for small figures, and for pieces of large figures.

To do the operation for all of the small figures, first the squirrels move right to left, one per row, with each counting the total size of all small figures with a leader in the row. Each total is divided by $\sqrt{n}$ and rounded up, with the result deposited in the leftmost column. This the minimum number of squirrels required if each visits no more than $\sqrt{n}$ pixels while labeling the small figures. For the entire image this could require nearly $2\sqrt{n}$ squirrels. We therefore require each squirrel to do the work of 2. If, for example, the first row had a total of $4.5\sqrt{n}$, then it was converted into a 5, so the first 2 squirrels will work solely on that row, and the third will work on that row and the next row with a nonzero value. Within a row the figure sizes may not divide evenly by $\sqrt{n}$, so the first squirrel does the first set of figures that add up to at least $2\sqrt{n}$, the second squirrel takes the next figure through the set of figures that add up to at least $4\sqrt{n}$, and the third squirrel takes the remaining ones (it is possible that there are none remaining). No squirrel works on more than 1 small figure more than its share, and since no small figure has size more than $\sqrt{n}$, no squirrel works on more than $3\sqrt{n}$ pixels. Hence the total time to complete the operation on all small figures is $\Theta(\sqrt{n}\,)$.

For the large figures the reduction operation will be described, with the broadcast being an approximate reversal of this. As before we assign squirrels to breakpoints, and the squirrel assigned to the breakpoint of rank $i\sqrt{n}$ will do the reduction over all pixels of rank $i\sqrt{n}$ through $(i+1)\sqrt{n}\ -1$, i.e., until the next breakpoint. A slight difficulty, however, is that these pixels may not be contiguous. For example, even if the entire image is a single figure, two pixels with consecutive ranks can be quite far from each other because of the jumps in the z-ordering (see Figure 2). However, a slight modification can remedy this. Given two pixels at positions p and q, the set of pixels with z-orderings between theirs form at most 2 convex regions C_1, C_2 that can easily be determined (see Figure 3). If p is a breakpoint of some figure F and q is the next breakpoint of F, then all of the points in F with ranks between the ranks of p and q lie in $C = C_1 \cup C_2$. Further, $F \cap C$ forms a collection of subfigures, each of which touches the boundary of C. Therefore a squirrel can start at p, follow the boundary of C, and whenever a new pixel of F is encountered start a depth-first search of that subfigure. The boundary of C has length $O(\sqrt{n}\,)$, and the total number of pixels examined in the depth-first search is $\leq \sqrt{n}$, so the total time for the squirrel to determine the reduction of its piece is $\Theta(\sqrt{n}\,)$. Then the squirrels can congregate in the middle to determine the reductions over entire figures and return the results to the leaders. Note that the number of pixels in C may be far larger than $\sqrt{n}$, so the squirrel could not simply traverse all of C. $\Box$

There is a slight complication in the above, in that several squirrels may have paths that overlap, and only a fixed number are allowed to occupy a position at any one time. However, simple routing control mechanisms can guarantee that all of the traversals can be completed in $O(\sqrt{n}\,)$ time.

Using reduction one can find the area and perimeter of each figure, and its bounding box, where the *(iso-oriented) bounding box* of F, denoted $\mathrm{box}(F)$, has x-extent equal to the x-extent of the pixels in F (i.e., the smallest to largest x-coordinates of pixels in F), and its y-extent is the y-extent of the pixels in F. A broadcast is used to move all of these values to all pixels in F.

A figure F is *contained in* figure G if every path from F to the edge of the image contains a pixel of G. Whenever two pixels of different colors are adjacent one corresponds to a figure containing the other (unless both figures are adjacent to the boundary), but it cannot be determined which is which without some global information. Fortunately this is simple to determine since a figure F is contained in an adjacent figure G iff $\mathrm{box}(F)$ is contained in $\mathrm{box}(G)$. Note that if figures are not adjacent then it can be that $\mathrm{box}(F) \subset \mathrm{box}(G)$

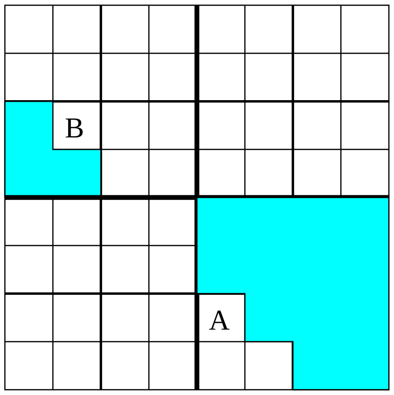


Figure 3: All locations with z-ordering between those of A and B

without G containing F. For example, if G is shaped like a U then F can be a dot inside it without being properly contained. However, for adjacent figures this cannot occur. Note that a figure has only one containing figure that is adjacent to it. The *nesting level* of a figure is the number of figures that contain it.

Proposition 4 *Given a strongly labeled image of size n, for each figure one can determine if it is contained in any others, obtain the label of the smallest container, and determine its nesting level, in $\Theta(n/s)$ time using peak energy s, for $1 \leq s \leq \sqrt{n}$. Further, for black/white images, in the same time/energy bounds each figure can determine the smallest container of the same color.*

Proof: For each pixel p on the boundary of a figure determine if an adjacent pixel in the figure of the opposite color is part of a figure with a bounding box containing p's, and if so retain that figure's label, while otherwise just retain an empty label. This label is that of the smallest container of a different color. To find the smallest container of the same color in a black/white image, now each boundary pixel of F adjacent to the containing figure G of the opposite color acquires the label of G's container, which is the smallest containing figure of F of the same color. Note that for arbitrarily many colors the closest enclosing figure of the same color may be many levels away.

To determine nesting level, a left-right scan can be used, adding one every time a transition is made from a figure to one it contains, and subtracting one when the opposite occurs. □

Given an image, suppose each pixel has a (possibly empty) label, not necessarily a label of a figure. The *closest similar point problem* is to find, for every pixel with nonempty label a closest one of the same label; the *closest black point problem* is to find, for every pixel, a closest black one; and the *closest differing point problem* is to find a closest one with a different non-empty label.

Theorem 5 *Using peak energy s, $1 \leq s \leq \sqrt{n}$, in $\Theta(n/s)$ time the closest black point problem, and the closest differing point problem, can be solved for the ℓ_1 and ℓ_∞ metrics.*

Proof: For the closest black point problem using the ℓ_1 metric, squirrels perform a right-left sweep in each row, leaving, at each position, the location of the most recently encountered (hence closest) black pixel. Then a similar left-right sweep is performed, where the closer of the black pixels in either direction is left at each pixel. Note that for any point p, either the closest black point is in the same column, or is one of the points recorded in its column (including the points recorded at p itself). Now vertical sweeps are done in each direction. Suppose an upward sweep is being done. The squirrel remembers the location of the closest black pixel known so far. At each position q, it compares the distance to the pixel it is carrying versus the distance to the black pixel's location stored at q in the horizontal sweeps, and it keeps the location of the closer of these two, proceeding upwards. At each step, when it arrives at a pixel it is carrying the location of the closest black pixel with vertical coordinate no larger than the coordinate it is currently at. A similar downward sweep is also done, at which time the correct value is stored at each location. Modifications for the ℓ_∞ metric are quite simple, and to modify for the closest differing pixel problem, note that the squirrel merely needs to keep track of the closest labeled point, and the closest one of a different label. $\square$

The closest similar point problem is difficult in that there can be images with $n/2$ labels where each occurs exactly twice, in which case the problem is essentially the same as sorting, requiring $\Omega(n^{3/2})$ total energy. However, when the only labels are black or white the problem becomes considerably easier.

Theorem 6 *Using peak energy s, $1 \leq s \leq \sqrt{n}$, in $\Theta(n/s)$ time the closest similar point problem can be solved for a black/white labeled image, where the metric is ℓ_1, ℓ_2, or ℓ_∞.*

Proof: For the ℓ_1 and ℓ_∞ metrics the problem was solved in Theorem 5. For the l_2 norm a somewhat more complex algorithm is used, closely following that in [11]. Figure 4 helps illustrate the approach. Suppose p is a black pixel within the dark subsquare, and suppose the closest black point has been found within the union of the subsquare and the horizontal and vertical bands (the lighter gray regions). The only way there might be a closer black pixel is if p is closer to a corner c than it is to any black pixel found so far, for in this case there might be a black pixel q in the white quadrant corresponding to c that is the closest one to p. Call p a *special point* if it satisfies this criterion. An important fact is that there are at most 2 points within the square which are closer to c than to any point found so far. If there were 3 or more such points, one would be closer to another than to the corner (see [11]). Thus in total there are at most 8 special points for which white regions need to be considered.

To start the process, each squirrel is assigned to a subsquare of size n/s. These are the black subsquares in Figure 4. In linear time, for each black point it locates the nearest black, if any, in the square. (It can do this by, say, simulating the recursive approach described for the entire image.) At this stage the only points which are not guaranteed to have found the closest are those points which are the leftmost or rightmost within a row, or highest or lowest in a column, of the subsquare. The only points within the banded regions which might be closest to them are the leftmost ones in each row in the banded region to the right of the square, the rightmost ones in each row in the banded region to the left of the square, and similarly for the vertical banded region. Row- and column-wise sweeps as in Theorem 5 can be used to simultaneously find the appropriate banded region points for all subsquares, and finding the points within the subsquare is similar. There are at most $\Theta(\sqrt{n/s})$ points within the square that have to consider at most $\Theta(\sqrt{n/s})$ points within the banded region, so simple comparisons of all of the inside points with all of the outside ones can be done in $\Theta(n/s)$ time. Then the special points are located. A traversal along the row corresponding to the top row of the square is performed, where at each column the distance from the special point to the lowest black pixel above that row is computed, and if this is closer than any point found so far then it is kept. Similar operations are performed along the horizontal and vertical bounding lines in all directions. The traversal takes $\Theta(\sqrt{n})$ time, and when all of the traversals are completed the closest black neighbor of each black pixel has been found. Then the same algorithm is applied to the white pixels, locating the closest white. $\square$

Note that the above approach does not directly solve the nearest black point problem for the ℓ_2 metric because there may be more than 8 special points. For example, there may be no black pixels in the square or banded regions, and hence every pixel in the square is special.

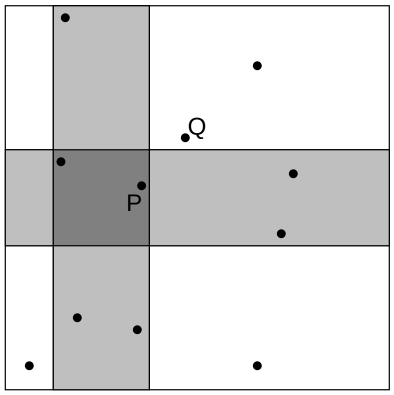


Figure 4: The closest point to P is Q

3 Graph Problems

3.1 Adjacency Matrix Input

General matrix operations are not amenable to peak energy reduction without increasing the time. For example, it appears that multiplying $\sqrt{n} \times \sqrt{n}$ matrices requires total energy $\Omega(n^{3/2})$ on a mesh, despite serial algorithms such as Strassen's which reduce the serial energy to $o(n^{3/2})$. However, operations involving adjacency matrices are often simpler. For example, some serial and parallel graph algorithms involve steps which compress the matrix, merging entries together and reducing the size of the remaining problem. Unfortunately, squirrels cannot compress the matrix without taking $\Omega(n^{3/2}/s)$ time — consider, for example, compressing a matrix where every other row and column has been eliminated. Thus energy efficient algorithms must rely on extracting subsets of information and moving them.

Theorem 7 *Given the $\sqrt{n} \times \sqrt{n}$ weighted adjacency matrix of an undirected graph, the connected components and a minimal spanning forest can be determined in $\Theta((n \log n)/s)$ time and peak energy s, for $1 \leq s \leq \sqrt{n} \log n$*

Proof: For $1 \leq s \leq \sqrt{n}$ a standard iterative method is used. At each stage, each array position starts with the weight of the edge, if any, and the labels of the two vertices. Then a horizontal sweep is used to discover, for each vertex v, the smallest edge in its row corresponding to a vertex with a label different from v's. These $\sqrt{n}$ edges are transported to a subsquare of size $\sqrt{n}$, where the minimal spanning forest of them is constructed and the vertices labeled to reflect the connected components. The squirrels do a vertical sweep on each column, and a horizontal one on each row, to propagate the new vertex labels and mark which edges were used. Then the next stage begins.

Each stage reduces the number of components at least by a factor of two, so at most $\lceil \log_4 n \rceil$ stages are needed. The sweeps take $\Theta(n/s)$ time, and the subsquare calculations take $\Theta(n^{3/4}/s)$ time since the problem can be solved in $\Theta(n^{1/4})$ time by a standard mesh computer of size $\sqrt{n}$ [17]. Thus the time is as claimed for $s \leq \sqrt{n}$.

While the above was described in a manner natural for an adjacency matrix, note that a different approach could have been taken, namely to subdivide the matrix into subsquares, as was used in images, solve the problem within each subsquare, and then recursively merge results. This approach will work because no matter how a graph $G = (V, E)$ has its edges subdivided, $E = E_1 \cup \ldots \cup E_m$, a minimal spanning forest of G can be formed from the union of the minimal spanning forests of the subgraphs $G_i = (V, E_i)$. For each of the subsquares, the number of vertices present is linear in the edgelength, as was true for the image data, so the data movement has the same order of magnitude. Further, these observations show that the approach in Theorem 1 can be used to extend to $\sqrt{n} \le s \le \sqrt{n} \log n$. $\square$

For a graph with $|E|$ edges one can use the same basic approach used initially and solve the problem in no more than $|E|/\sqrt{n}$ iterations, by reassigning squirrels working on vertices where all the edges have already been used. Thus graphs of bounded degree, or of bounded average degree (such as planar graphs) can be solved in time linear in the number of vertices by $\sqrt{n}$ squirrels. For general graphs it is unknown if the worst case can be improved to remove the logarithmic factor in Theorem 7.

The basic approach used above consists of stages of collecting some information from the adjacency matrix, moving it to a smaller region to solve a graph problem with edges as input, and then moving information back to the adjacency matrix. This approach can be used for several other problems, such as taking an arbitrary spanning tree and directing it, i.e., selecting a root and having every vertex point to its parent. Given a directed tree, one can define tree reduction operations such as having every vertex know the reduction of all values in its subtree, or in the path from the root to it. These reductions can be used to determine sizes of subtrees, depth, height, position in a depth-first or breadth-first ordering, etc. All can be done with in $\Theta(n/s)$ time using peak energy s, $1 \le s \le \sqrt{n}$, once a spanning forest has been found [1, 12, 18]. The proofs are omitted as they closely follow those in these references.

Theorem 8 *Given an adjacency matrix of an unordered graph G with $\sqrt{n}$ vertices, and a spanning forest for G, then, using peak energy s, $1 \le s \le \sqrt{n}$, in $\Theta(n/s)$ time one can*

- *Decide if G is bipartite.*
- *Determine the cyclic index of G.*
- *Determine all bridge edges of G (and hence decide if G is biconnected).*
- *Determine all articulation vertices of G.*

$\square$

3.2 Edge List Input

As has been noted above, there are many algorithms for edge data which can be done in $\Theta(\sqrt{n})$ time, but $\Theta(n)$ peak power and $\Theta(n^{3/2})$ total power, on a standard mesh-connected computer. Further, for problems such as component labeling, it is easy to see that the total power is a lower bound, no matter what time or peak power is used. For example, there can be $2n/3$ vertices and $n/3$ components of size 3 containing 2 edges, where each pair of edges is $\Theta(\sqrt{n})$ apart. However, the situation can be improved if there are fewer vertices or components, much like the $\sqrt{n}$ vertices inherent when an adjacency matrix is used.

Theorem 9 *Given a graph $G = (V, E)$ with n edges stored one per processor, the connected components and a minimal spanning forest of G can be determined in time $\Theta(n\sqrt{V}/s)$ using peak energy s, $1 \le s \le \sqrt{|V|n}$.*

Proof: Within squares of size $|V|$ simulate a standard mesh algorithm, determining the minimal spanning forest using only the edges in the square. Note that some vertices may have no edges in the square. Then squares are merged together in a series of stages, each stage combining 4 subsquares at a time. The merger involves moving the $\leq |V|$ edges together into a subsquare and using a basic mesh algorithm algorithm to reduce down to the minimal spanning forest of these, resulting in no more than $|V|$ edges.

The first time squares are combined, at most $|V|$ edges are moved a distance of $\sqrt{|V|}$, taking $|V|^{3/2}$ total energy per square and $n\sqrt{|V|}/s$ time since $s|V|/n$ squirrels are assigned to the square. At each subsequent stage the distance the edges are moved increases by a factor of 2, without increasing the number of edges, and the number of squirrels moving them increases by a factor of 4, so the total time decreases by a factor of 2. Thus the time is dominated by the first stage. The initial stage, using a basic mesh algorithm in squares of size $|V|$, can be completed with $\Theta(|V|^{3/2})$ total energy and $\Theta(|V|^{3/2}/s)$ time by $1 \leq s \leq |V|$ squirrels [17]. Here $s = r|V|/n$, so the time is $\Theta(n\sqrt{|V|}/s)$, i.e., the energy and time are the same as the initial move to combine squares. The total energy over all of the initial squares is $\Theta(n\sqrt{|V|}\,)$. $\square$

One can apply the above technique to compute broadcasts and reductions over components once they have been labeled. If there are C components then smaller initial squares, of size C, can be used. The time will be reduced to $\Theta(n\sqrt{C}/s)$, and the total energy will be reduced to $\Theta(n\sqrt{C}\,)$. These time and energy bounds can similarly be obtained for the problems mentioned in Theorem 8. Further, it is easy to see that this time and energy are optimal: suppose in each initial square there is exactly one value for each component. Since the components are independent, the total time and energy to perform reductions over all components is C times the value for a single component. The single component values can be viewed as being in a square lattice of size n/C, where the distance between neighbors is $\sqrt{C}$. Thus to perform reduction over any component requires energy proportional to $\sqrt{C}n/C = n/\sqrt{C}$, so for C components the total is proportional to $n\sqrt{C}$. Similarly, the time and power achieved in Theorem 9 is worst-case optimal.

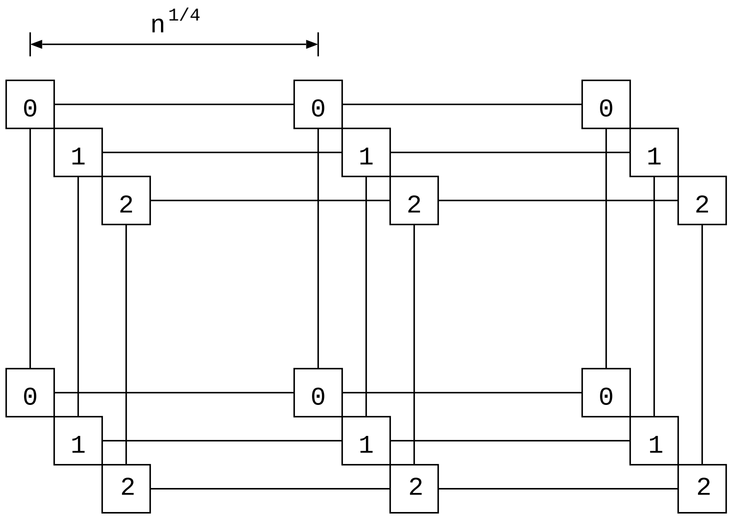
Note that V, and hence $|V|$, need not be known in advance. The algorithm can start assuming a small value, and when results from subsquares are merged together, if it is discovered that the guess was too small because there are too many different vertex labels present, then the squirrels in that square continue the algorithm for the larger size. Other squirrels may end up waiting for such a group, at which point they too will then proceed with the larger value. These cascades can occur several times until the true value $|V|$ is learned; however, the resulting time is asymptotically the same as if it had been known in advance.

4 Minimizing Total Energy Used by Any Single Processor

In some applications, such as sensor networks, an important consideration is the maximum energy used by any sensor. This is because the sensors are assumed to have their own, limited, power, as opposed to the externally supplied power which motivated this work.

Recall that the power used by a processor corresponds to the number of times it was visited by a squirrel. For the preceding algorithms, in most steps no processor is visited more then $\Theta\left(\frac{1}{n} \cdot (\text{Total Power})\right)$ times, i.e., no processor had power requirements more than the average. However, when data was collected and moved to a subsquare for processing by a standard mesh-connected computer algorithm then this was not true, since processors in the subsquare were on continuously during this step, far more than their share. For example, in the last stage of recursion in the image component algorithm for Theorem 1, the processors simulate a standard mesh algorithm taking $\Theta(n^{1/4})$ time, and hence expend $\Theta(n^{1/4})$ energy, but the average per processor for the entire algorithm is only $\Theta(\log n)$.

However, these power-intensive steps can be modified so that the average power per processor is still a



Numbers indicate when the processor is used to simulate the standard mesh algorithm

Figure 5: Expanding the simulating mesh

constant. To do so, the $n^{1/4} \times n^{1/4}$ submesh is expanded, as in Figure 5, so that the simulating processors are $n^{1/4}$ apart. Then a fixed number of steps of the simulation are performed, where each step now takes $\Theta(n^{1/4})$ time. After this, the location of the simulating processors are moved diagonally 1 step, as indicated in the figure, and then another fixed number of steps are simulated, and so forth. The number of steps is chosen so that the simulated algorithm is finished by the time the diagonal movement would place simulating processors on top of ones previously used.

It is easy to see that now no processor is used more than a constant number of times during the simulation, either for calculation or as part of a communication path. While the notion of grouping processors together and having only a few be active at one time has been used in sensor networks (e.g., [8, 16]), it is unusual to do this when it significantly increases the time. The time has increased by a factor of $n^{1/4}$, but this merely makes it equal to the time needed to move data to the subsquare. Thus it does not increase the total time by more than a constant multiple. Similar changes can be made concerning the use of the leftmost column in Theorem 3.

Summarizing, we have the following

Theorem 10 *Using the indicated changes, all of the preceding algorithms can be modified so that the peak power, total power, and time do not change by more than a constant multiple, and simultaneously each processor uses only* $\mathrm{O}\left(\frac{1}{n} \cdot (\text{Total Power})\right)$ *power.* □

5 Final Remarks

This paper has been concerned with minimizing peak power usage by mesh-connected computers. The algorithms herein were only given for 2-dimensional meshes, especially since some of the data formats are naturally 2-dimensional, but one can use similar approaches to develop power reducing algorithms for higher dimensional meshes. However, as the dimension increases there is a smaller range in which to lower peak power without increasing the time required. For example, in a 3-dimensional mesh of n processors, summing values from each processor can be done in $\Theta(n^{1/3})$ time, and any algorithm which achieves this minimal time must have a peak power of $\Omega(n^{2/3})$. In contrast, in 2 dimensions the minimal time is $\Theta(n^{1/2})$, which can be achieved with a peak power of $\Theta(n^{1/2})$.

A more optimistic viewpoint is that for the same energy bounds there are problems that a 3-d mesh can solve faster than a 2-d one, just as 2-d meshes can be superior to 1-d meshes. Sorting is one such example. For most of the problems in this paper, however, that is not possible since the algorithms provided work-optimal tradeoffs compared to serial algorithms. For some problems it isn't clear what effect the dimension has. For example, to find the median, the optimal time-energy tradeoffs are unknown for arbitrary dimensions.

In general, the more structured the input the easier it is to reduce the power. For example, adjacency graphs yield faster algorithms than do unordered edges. Part of the explanation for this is that less global rearrangement of the data is needed, an operation which is quite power-intensive. In some cases one might need to do an initial power-intensive operation with restricted peak power, in which case the time will increase. However, there might be efficient algorithms for properly organized data. For example, some geometry problems on point data can be solved significantly faster if the points have been sorted by x-coordinate. Thus if one is solving a sequence of such problems it may be useful to organize the data initially and view the organizational time as being amortized over subsequent operations.

It is interesting to note that several of the algorithms use a fixed pattern of energy use, i.e., the time at which a processor is active is independent of the data. The messages passed, of course, do depend on the data. One exception was the depth-first search used by individual squirrels to label figures in a subsquare at the start of the algorithm in Theorem 1. They could have used the fixed activation pattern employed by the subsequent stages of the labeling process, but an extra logarithmic factor would have been introduced. It is unclear what the optimal time for a single squirrel is if its pattern of motion must be independent of the data.

References

[1] Atallah, M.J. and Hambrusch, S.E., "Solving tree problems on a mesh-connected processor array", *Infor. and Control* 69 (1986), 168–187.

[2] Atallah, M.J. and Kosaraju, S.R., "Graph problems on a mesh-connected processor array", *J. ACM* 31 (1984), 649–667.

[3] Batcher, K.E., "The design of the Massively Parallel Processor", *IEEE Trans. Computers* C29 (1980), 836–840.

[4] BlueGene/L: http://www.research.ibm.com/bluegene/

[5] Blum, M. and Hewitt, C., "Automata on a 2-dimensional tape", *Proc. 8th IEEE Conf. SWAT* (1967), 155–160.

[6] Blum, M. and Sakoda, W.J., "On the capability of finite automata in 2 and 3 dimensional space", *Proc. 18th Symp. Foundations of Computer Science*, 1977, 147–161.

[7] Brown, C., "Algorithm yields ultra-low-power image sensor", *EE Times*, December 12, 2005.

[8] Chen, B., Jamieson, K., Balakrishnan, Morris, R., "Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks", 7th MOBICOM, 2001.

[9] Leighton, F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, 1992, Morgan Kaufmann.

[10] Mattson, T. and Henry, G., “An overview of the Intel TFLOPS supercomputer”, *Intel Tech. J.* 2 (1968).

[11] Miller, R. and Stout, Q.F., “Mesh computer algorithms for computational geometry”, *IEEE Trans. Computers* 38 (1989), 321–340.

[12] Miller, R. and Stout, Q.F., *Parallel Algorithms for Regular Architectures: Meshes and Pyramids*, 1996, MIT Press.

[13] Nassimi, D. and Sahni, S., “Finding connected components and connected ones on a mesh-connected parallel computer”, *SIAM J. Computing* 9 (1980), 744–757.

[14] Rabin, M., “Maze threading automata”, unpublished lecture, 1967.

[15] Shah, A.N., “Pebble automata on arrays”, *Comp. Graphics and Image Proc.* 1974, 236–246.

[16] Singh, M., Bakshi, A., and Prasanna, V.K., “Constructing topographical maps in networked sensor systems”, *Proc. ASWAN 2004*.

[17] Stout, Q.F., “Optimal component labeling algorithms for mesh-connected computers and VLSI”, *Abstracts AMS* 5 (1984), 148.

[18] Stout, Q.F. “Tree-based graph algorithms for some parallel computers”, *Proc. 1985 Int’l. Conf. Parallel Proc.* (1985), 727–730.

[19] Thompson, C.D. and Kung, H.T., “Sorting on a mesh-connected parallel computer”, *Comm. ACM* 20 (1977), 263–271.

[20] Tilera Corporation, “TILE64 Processor Produce Brief” (2007),
www.tilera.com/pdf/ProBrief_Tile64_Web.pdf

[21] Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., and Catalyurek, U., “A scalable distributed parallel breadth-first search algorithm on BlueGene/L”, *Proc. SC—05* (Supercomputing 2005).