Complexity of Sub-Bus Mesh Computations*

Anne Condon

Department of Computer Science
University of Wisconsin
Madison, WI 53706

Richard E. Ladner
Jordan Lampe
Rakesh Sinha
Department of Comp. Sci. and Eng.
University of Washington
Seattle, WA 98195

September 20, 1994

Abstract

We investigate the time complexity of several fundamental problems on the sub-bus mesh parallel computer with p processors. The problems include computing the PARITY and MAJORITY of p bits, the SUM of p numbers of length $O(\log p)$ and the MINIMUM of p numbers. It is shown that in one dimension the time to compute any of these problems is $\Theta(\log p)$. In two dimensions the time to compute any of PARITY, MAJORITY, and SUM is $\Theta(\frac{\log p}{\log \log p})$. It was previously shown that the time to compute MINIMUM in two dimensions is $\Theta(\log \log p)$ [21, 31].

*This is a revision of University of Washington Technical Report No. 93-10-02. This paper will appear in *SIAM Journal on Computing*. Condon's research was supported by NSF, grant numbers CCR-9100886 and CCR-9257241. Ladner's and Lampe's research was supported by NSF, grant number CCR-9108314. Part of Ladner's research was done while visiting Victoria University of Wellington, New Zealand. Sinha's research was supported by NSF/DARPA grant number CCR-8907960 and NSF grant number CCR-8858799.

1 Introduction

A sub-bus mesh computer is a single-instruction multiple-data (SIMD) two-dimensional array of processors, where processors can broadcast data vertically or horizontally on segmented busses. To segment a bus, some of the processors on the bus are active while others are inactive. Each active processor can broadcast on the bus to all processors up to the next active processor. Thus, all the intervening inactive processors receive the data that has been broadcast. The sub-bus mesh computer architecture has been implemented on the commercially available MasPar MP-1 [5].

Typically, there are p processors in a $\sqrt{p} \times \sqrt{p}$ two-dimensional array, where $\sqrt{p}$ is a power of two. We will also consider one-dimensional meshes. For each of the problems we will consider, we assume there are p inputs, distributed one per processor.

The purpose of this paper is to present upper and lower bounds on the time to compute several fundamental functions including the PARITY and MAJORITY of p bits, the SUM of p numbers of length $O(\log p)$, and the MINIMUM of p numbers. Reisis and Prasanna Kumar appear to be the first to consider efficient algorithms for the sub-bus mesh architecture [26]. To our knowledge this is the first paper to prove extensive results on the power and limitations of the sub-bus computer architecture.

1.1 Results

There are simple algorithms for computing the Boolean OR and AND in constant time on a one- or two-dimensional sub-bus mesh computer [26]. Two other natural Boolean functions are PARITY and MAJORITY. We show that PARITY and MAJORITY are computable in time $\Theta(\log p)$ on a one-dimensional sub-bus mesh. The proofs of the lower bounds use a new technique for bounding the amount of information about the input which can be distributed on a one-dimensional mesh. The proof of the lower bound for MAJORITY can be used to show lower bounds for some other symmetric Boolean functions. We also consider the problem of computing the MINIMUM on a one-dimensional sub-bus mesh, and show that MINIMUM is computable in $\Theta(\log p)$ time. The lower bound for MINIMUM on the one-dimensional sub-bus mesh uses the same technique as the lower bound for PARITY.

We then show that PARITY and MAJORITY are computable in time $\Theta(\frac{\log p}{\log\log p})$ on a two-dimensional sub-bus mesh computer. The lower bounds follow from the fact that a CRCW PRAM can simulate a sub-bus mesh computer to within a constant factor of the time and within a polynomial number of processors. Thus, the CRCW PRAM lower bounds on PARITY and MAJORITY [4] apply to the sub-bus mesh computer. The upper bounds follow from an algorithm for SUM, the sum of p numbers of length $O(\log p)$, which runs in time $O(\frac{\log p}{\log\log p})$. The SUM algorithm is non-trivial, using mixed radix arithmetic, the Chinese remainder theorem and recursion to achieve the result. The obvious algorithm for SUM takes time $\Theta(\log p)$. The two-dimensional bound of $\Omega(\frac{\log p}{\log\log p})$ for SUM on the CRCW PRAM follows from the lower bound on PARITY.

1.2 Related Results

The mesh or array parallel computer architecture has been investigated for a number of years, with numerous articles on its many variants [13, 17, 18, 22, 21, 26, 29]. The sub-bus mesh architecture was first investigated by Reisis and Prasanna Kumar where they gave constant time algorithms for the OR of p bits and the MINIMUM of $\sqrt{p}$ numbers (all on one row), and an $O(\log p)$ algorithm for combining p data items with an associative operator [26]. Two variants of the mesh computer are closely related to the sub-bus mesh. First, there is the full-bus mesh where processors can broadcast vertically or horizontally, but on a vertical (horizontal) broadcast at most one processor per column (row) can be active. The MPP of Goodyear and NASA is an example of a full-bus two-dimensional mesh computer [3]. Full-bus meshes are generally less powerful than sub-bus meshes. Both PARITY and MINIMUM require $\Omega(p^\alpha)$ time for some $\alpha > 0$ on full-bus meshes [2, 25].

Second, there is the reconfigurable mesh, which allows the topology of the mesh to be changed by the executing program [18]. Several prototype, but no commercial, reconfigurable mesh computers have been built. PARITY can be computed in constant time on the reconfigurable, two-dimensional mesh [18]. Thus, our results demonstrate that the sub-bus mesh computer architecture is strictly more powerful than the full-bus mesh computer architecture, but strictly less powerful than the reconfigurable mesh computer. In other work on the PARITY function, MacKenzie [19] independently obtained a lower bound of $\Omega(\log p/k)$ for computing PARITY on a restricted $p \times k$ reconfigurable mesh model, which is exactly our one-dimensional sub-bus mesh model for $k = 1$. However, he did not extend this to other symmetric functions.

The SUM function has also been previously studied on the reconfigurable mesh. Nakano [23] and Nakano, Masuzawa and Tokura [24] developed algorithms for SUM on the reconfigurable mesh that also use Chinese remaindering. Their results do not apply directly to the sub-bus mesh architecture. MINIMUM has also been previously studied on the two-dimensional reconfigurable mesh, and the techniques used can be applied directly to show that MINIMUM can be computed in $\Theta(\log\log p)$ time on the two-dimensional sub-bus mesh. From the work of Hao, MacKenzie and Stout [12], a lower bound of $\Omega(\log\log p)$ is obtained for computing MINIMUM on a two-dimensional sub-bus mesh. Their proof is based on a PRAM simulation of the mesh model, and applies a result of Fich *et al* [11] in which an equivalent lower bound is proved for the CRCW PRAM. However, this proof requires that the inputs be very large. Another lower bound of $\Omega(\log\log p)$ for computing MINIMUM on the two-dimensional sub-bus mesh follows from the general lower bound for the parallel comparison model of Valiant [31] and applies to comparison-based algorithms. A matching upper bound is due to Miller *et al* [21], and is basically an implementation of the parallel MINIMUM algorithm of Valiant [31].

The parallel random access machine (PRAM) is probably the most well studied theoretical model of parallel computation. There are a number of variants of the PRAM depending on whether reads or writes to the same memory location can be done concurrently. The variant most closely related to the sub-bus mesh is the CRCW PRAM (concurrent read/concurrent write PRAM) (see [14] for an introduction to the PRAM model). In this

version more than one processor can read or write the same memory location at the same time. A simultaneous write can be resolved in a number of ways. The ability of the sub-bus mesh to broadcast on segments of the bus is very much like a combination of a concurrent read and a concurrent write. Those processors which are actively broadcasting are executing a concurrent write while those which are inactive are executing a concurrent read. Indeed the sub-bus mesh can compute OR in constant time just as a CRCW PRAM can. We show that CRCW PRAM can simulate a sub-bus mesh computer to within a constant factor of the time. This simulation immediately implies that lower bounds for the CRCW PRAM are also lower bounds for sub-bus mesh. Interestingly, some CRCW PRAM algorithms can be translated into sub-bus mesh algorithms. For example, the constant time OR algorithm and Valiant's MINIMUM algorithm can be implemented on the sub-bus mesh. By contrast, some CRCW PRAM algorithms, such as the constant time CRCW PRAM MINIMUM algorithm of Fich, Radge, and Wigderson [10], appear to be impossible to implement on the sub-bus mesh. More generally, the CRCW PRAM is strictly more powerful than the sub-bus mesh regardless of the number of dimensions. This is because problems like SORT require time $\Omega(p^\alpha)$ on mesh computers, but can be done in time $O(\log p)$ on a PRAM.

The sub-bus model is also incomparable with the EREW (exclusive read, exclusive write) PRAM model. To see this, note that computing the OR of p bits requires $\Omega(\log p)$ time on a CREW PRAM [9], whereas the sub-bus mesh can compute OR in constant time. By contrast, p integers can be sorted in $O(\log p)$ time on an EREW PRAM with p processors [1, 7, 16], but, by a simple bisection bandwidth argument, this task requires time $\Omega(p^\alpha)$ on mesh computers [30].

1.3 Organization of the Paper

In section 2 we present our model of the sub-bus mesh computer. In section 3 we prove our upper and lower bounds for the one-dimensional sub-bus mesh computer. In section 4 we give two-dimensional algorithms for PARITY and SUM. In section 5 we give our simulation of a two-dimensional sub-bus mesh by a CRCW PRAM thereby yielding our two-dimensional lower bounds for PARITY, MAJORITY, and SUM. In section 6 we present two-dimensional upper and lower bounds for MINIMUM. Finally, we have our conclusions in section 7.

2 Sub-bus Mesh Computer Model

For the purposes of this paper we present a simple version of the sub-bus mesh computer architecture. Actual machines have a richer organization.

The sub-bus architecture can be easily explained for the one-dimensional mesh or linear array of processors. There are p processors, numbered consecutively 0 to $p-1$ on a circle. Processor 0 is the *front-end* which runs the parallel program. Each processor is a RAM with its own memory which is referenced using *plural* variables. In addition, there are *singular* variables for which there is only one copy which is stored at the front-end processor. There

is a special plural variable `PID` which always holds the processor's number. Processor operations include direct and indirect Boolean operations, arithmetic operations, shifts, and comparisons. In addition, the front-end can perform normal branching operations and issue *parallel instructions*.

A parallel instruction issued by the front-end has the form "**if** <*condition*> **then** <*statement*>." Each processor evaluates the condition, which can be any sequence of non-branching operations on plural or singular data which evaluates to a Boolean value. If the condition is true then the processor is said to be *active*, otherwise it is said to be *inactive*. Only the active processors execute the statement part of the instruction.

There are two kinds of statements, *local operations* and *segmented broadcasts*. A local operation is just a typical non-branching RAM operation executed on plural or singular data at each processor. A segmented broadcast has the form

`broadcast_`*direction*`[`*distance*`]`*.plural variable* $\leftarrow$ *plural variable*;

The *direction* can be either `left` or `right`. The variable *distance* must be singular. When an active processor i executes the instruction `broadcast_right[`d`].y` $\leftarrow$ `x` then the location `y` at the processors $(i+1) \bmod p, (i+2) \bmod p, ..., (i+j) \bmod p$ receive the value stored in location `x` of processor i, where processors $(i+1) \bmod p, (i+2) \bmod p, ..., (i+j-1) \bmod p$ are inactive and either $j = d$ or $j < d$ and processor $(i+j) \bmod p$ is active. The segmented broadcast to the left is similar. In either case, the circle is partitioned into non-overlapping segments. Each segment behaves like a sub-bus of the bus which includes all the processors. The MasPar MP-1 implements the segmented broadcast as `xnetc`. Table 1 describes the result of a segmented broadcast.

	`broadcast_right[2].y = x`							
`PID`	0	1	2	3	4	5	6	7
active	no	no	yes	yes	no	no	no	yes
`x`	a	b	c	d	e	f	g	h
`y`	h	h	*	c	d	d	*	*

Table 1. Demonstration of segmented broadcast. The * indicates that the value of `y` did not change because of the broadcast.

In two dimensions there are also p processors where p is a square number. The mesh processors are arranged in a $\sqrt{p} \times \sqrt{p}$ array. The coordinates of a mesh processor's number are stored in `PIDx` and `PIDy`. A mesh processor's number is stored in `PID` = `PIDy` * $\sqrt{p}$ + `PIDx`. The sub-busses go in four directions `up`, `down`, `right`, and `left`. Processor (x, y) is immediately up from processor $(x, (y+1) \bmod \sqrt{p})$ and immediately to the left of processor $((x+1) \bmod \sqrt{p}, y)$. So vertical busses go up and down, while horizontal busses go right and left all in a circular fashion. The front-end processor is processor $(0, 0)$.

2.1 Time of Sub-Bus Mesh Algorithms

For the purpose of analyzing our algorithms we consider time to be evaluated using the unit cost RAM criterion where the values operated upon must have length $O(\log p)$. Each

sequential operation by the front-end, each parallel operation used in evaluating the condition in a parallel instruction, and each statement of a parallel instruction costs 1 in our model. We do not charge for the broadcast of parallel instructions by the front-end to the mesh processors. We assume that cost is dominated by the cost of executing the parallel instruction.

As mentioned earlier, the RAM instructions include the usual direct and indirect Boolean operations, arithmetic operations, shifts, and comparison. In addition, we permit any fixed finite set of RAM instructions for our processors, provided each instruction can be implemented in uniform NC [8, 28], that is, each instruction can be built from $\log^{O(1)} p$ hardware and runs in time $\log^{O(1)} \log p$. The set of RAM operations is independent of p. Thus, the total hardware in the sub-bus mesh computer of p processors is $p \log^{O(1)} p$. The running time of $\log^{O(1)} \log p$ per RAM instruction is fast enough to be considered to be constant time in the mesh of processors.

For the purpose of proving our lower bounds we allow our model to be even more general. We do not restrict the length of the values operated on and do not restrict the RAM operations in any way. There is one exception. The two-dimensional lower bound for MINIMUM is done in the so-called "comparison model" where the only RAM operations allowed on input values are comparison, copy, and broadcast. Thus, with one exception, the lower bounds reflect the cost of computing functions due to the sub-bus mesh architecture, not any limitations on the individual processors. The two-dimensional lower bound for MINIMUM is still quite general, but is limited to the comparison model of the sub-bus mesh computer.

2.2 Examples of Sub-Bus Mesh Algorithms

Below we give two examples of sub-bus mesh algorithms both of which will be building blocks in subsequent algorithms. Both examples can be found in the paper by Reisis and Prasanna Kumar [26]. In our terminology if `x` is a plural variable then we indicate its value at processor i by $\mathtt{x}_i$ or at processor (i, j) by $\mathtt{x}_{i,j}$.

Our first program computes the OR in constant time.

```
CONSTANT-TIME-OR on one-dimensional mesh
input: plural boolean x
output: OR of all values x_i in plural variable y
begin
   y ← false
   if x = true then
      broadcast_left[p].y ← true
end
```

In CONSTANT-TIME-OR, if any of the values $\mathtt{x}_i$ are `true`, then one or more of the processors will make sure to broadcast `true` into all processors' `y`'s. However, if *none* of

the $\mathtt{x}_i$ bits are `true`, then no processor will run the broadcast step, and so all the $\mathtt{y}_i$'s will remain `false`. Clearly, using DeMorgan's law AND can also be computed in constant time.

Our second program computes the MINIMUM of $\sqrt{p}$ values in constant time on a two-dimensional $\sqrt{p} \times \sqrt{p}$ mesh.

CONSTANT-TIME-MINIMUM of $\sqrt{p}$ values on a two-dimensional mesh
input: plural integer/real variables $\mathtt{x}_{0,0}, \ldots, \mathtt{x}_{\sqrt{p}-1,0}$
output: MINIMUM value $\mathtt{x}_{i,0}$ in plural variable `y`
other: plural boolean `t`
begin
  if `PIDy = 0` **then**
    `broadcast_down[`$\sqrt{p}-1$`].x` $\leftarrow$ `x`
  if `PIDx = PIDy` **then**
    `broadcast_left[`$\sqrt{p}$`].y` $\leftarrow$ `x`
  `t` $\leftarrow$ `x > y`
  if `t` **then**
    `broadcast_up[`$\sqrt{p}-1$`].t` $\leftarrow$ `true`
  if not `t` **then**
    `broadcast_left[`$\sqrt{p}$`].y` $\leftarrow$ `x`
end

In CONSTANT-TIME-MINIMUM, the first two broadcasts have the effect of setting $\mathtt{x}_{i,j} = \mathtt{x}_{i,0}$ and $\mathtt{y}_{i,j} = \mathtt{x}_{j,0}$. The comparison $\mathtt{x}_{i,j} > \mathtt{y}_{i,j}$ is then equivalent to the comparison $\mathtt{x}_{i,0} > \mathtt{x}_{j,0}$. If such a comparison holds then $\mathtt{x}_{i,0}$ is not the minimum. The statement "`if t then...`" computes, in one step, the "or" of the outcomes of these comparisons. Thus, after the broadcast up, if $\mathtt{t}_{i,0} =$ `false` then $\mathtt{x}_{i,0}$ is the minimum. This minimum is then broadcast to the first row of the mesh.

3 One-Dimensional Bounds

In this section we give precise upper and lower bounds on the parallel time to compute PARITY, MINIMUM, MAJORITY and SUM on the one-dimensional sub-bus mesh computer.

3.1 Upper Bounds in One Dimension

As observed by Reisis and Prasanna Kumar [26], all our problems can be computed by a one-dimensional algorithm which works for any associative binary operator and runs in $O(\log p)$ time. Let $\oplus$ be any binary associative operation. The value REDUCE-$\oplus$(`x`) is $\mathtt{x}_0 \oplus \mathtt{x}_1 \oplus \ldots \oplus \mathtt{x}_{p-1}$ stored in processor 0. The following algorithm simply computes the expression for REDUCE-$\oplus$(`x`) as a balanced binary tree.

```
REDUCE-⊕ on one-dimensional mesh
input: plural variable x.
output: y_0 = x_0 ⊕ x_1 ⊕ ... ⊕ x_{p-1}
other: plural variable z, singular integer i
begin
   y ← x
   i ← 1
   while i < p do begin
      if PID mod i = 0 then
         broadcast_left[i].z ← y
      if PID mod 2i = 0 and PID + i < p then
         y ← y ⊕ z
      i ← i * 2
   endwhile
end
```

Both SUM and MINIMUM can be expressed as REDUCE-$\oplus$ operations where $\oplus$ is integer addition in the case of SUM and the minimum of two numbers in the case of MINIMUM. PARITY and MAJORITY are easily computable in constant time from SUM. Thus we have:

Theorem 3.1 *[26] On a one-dimensional sub-bus mesh with p processors, PARITY, MAJORITY, SUM, and MINIMUM can be computed in time* $O(\log p)$.

3.2 Lower Bounds in One Dimension

Our lower bounds for the one-dimensional sub-bus mesh computer are based on the limited communication bandwidth of this architecture. Thus, in proving our lower bounds, we use a simplified model, in which internal computations in a processor are "free" and only the time taken for communication is measured. It will be clear that any lower bound for this model applies also to the upper bound model.

As before, there are p processors, numbered $0, 1, \ldots, p-1$, connected by a circular sub-bus. The computation proceeds in rounds; we charge 1 time unit for a round. In each round of the computation, the processors first communicate and then perform arbitrary internal computation. The communication is controlled by the front-end, processor 0, just as in the upper bound model described in section 2. Once this is done, processors can do arbitrary internal computation that does not require communication. There is no bound on the length of values broadcast or computed by the processors.

An algorithm consists of both the algorithm that determines the sequence of communication instructions broadcast by processor 0, and the algorithms of processors $0, \ldots, p-1$ that determine the internal computations at each round. Since processor 0 can read any information on the bus that passes in either direction between processor $p-1$ and processor 1, the communication instructions may depend on this information.

Let f be a function with domain D^p. We say algorithm A computes f if for all $(x_0, x_1, \ldots, x_{p-1}) \in D^p$, if at the start of the algorithm each processor i has in its memory the value x_i, then at the end of the algorithm, every processor has in a special memory location the value $f(x_0, x_1, \ldots x_{p-1})$. The tuple $(x_0, x_1, \ldots, x_{p-1})$ is called the input. In all of the results of this section, we assume that $|D| \geq 2$.

Fix an input $\boldsymbol{x} = (x_0, x_1, \ldots, x_{p-1})$. Processor k's *view* on input $\boldsymbol{x}$ at time t is a sequence $k, x_k, (1, v_1), (2, v_2), \ldots, (t, v_t)$ where v_i is the value received by processor k at time i during the broadcast instruction. String v_i is a special symbol, say ϵ, if no value is received. We denote by $\text{View}_k(\boldsymbol{x}, t)$ the view of processor k at time t. For a fixed input $\boldsymbol{x}$, we say x_i is *unknown to* processor k at time t if $\text{View}_k(\boldsymbol{x}, t) = \text{View}_k(\boldsymbol{x}', t)$ for all $\boldsymbol{x}'$ that differs from $\boldsymbol{x}$ only at component i. Otherwise, we say x_i is known to processor k at time t.

Our main result is the following:

Theorem 3.2 *On a one-dimensional sub-bus mesh with p processors and for any algorithm A, there exists an input $\boldsymbol{x}$ such that for some $i, 1 \leq i \leq p-1$, x_i is unknown to processor 0 at time $\log p - 1$.*

This result is true, regardless of what function is being computed by A, as long as the domain size $|D| \geq 2$. Hence, the result immediately yields a lower bound of $\log p$ for the time to compute functions f with the property that for any $(x_0, x_1, \ldots, x_{p-1}) \in D^p$ and any $i, 0 \leq i \leq p-1$, there is some $x_i' \in D$ such that

$$f(x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{p-1}) \neq f(x_0, x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_{p-1}).$$

Clearly PARITY is an example of such a function, where $D = \{0, 1\}$, and so theorem 3.2 implies a lower bound of $\log p$ for PARITY. Also, the MINIMUM function over the integers is an example of such a function, so again theorem 3.2 implies a lower bound of $\log p$ for computing MINIMUM.

We now describe informally the ideas in the proof of theorem 3.2. Note that, for all inputs $\boldsymbol{x}$ and processors k, if $i \neq k$ then x_i is unknown to k at time 0. Consider a processor i at the first round. We consider two possibilities. The first is that i is inactive at round 1, regardless of its input value x_i. This is good since then, for any processor $k \neq i$, x_i is still unknown to processor k at time 1.

The other possibility is that i is "potentially active"; that is, i is active on at least one possible value of its input. Then, unfortunately, at the end of round 1, x_i may be known to some, and possibly all, other processors. We can use this to our advantage, however, by setting i's input x_i to force i to be active. Then, the broadcast of processor i will block any other broadcast which might have otherwise sent information through i.

For the purpose of this informal discussion, suppose that at round 1, all processors are potentially active. Our strategy in this case will be to fix the values of alternate processors, in order to force them to be active. These fixed values determine a partial assignment $\alpha \in (D \cup \{*\})^p$, and partition the processors into *fixed* and *free* processors. On any input $\boldsymbol{x}$ consistent with the partial assignment α, the broadcasts of the fixed, active processors

block the free processors from revealing any information about their values to too many processors.

In general, for any $t, 0 \leq t \leq \log p - 1$, we will define a partial assignment α which fixes the input at all but $\lfloor (p-1)/2^t \rfloor$ free processors. On any input $\boldsymbol{x}$ consistent with the partial assignment α, the input x_i of a free processor i will be known only to a set of contiguous processors containing i at time t.

We now state and prove the main lemma leading to the proof of theorem 3.2.

Lemma 3.1 *Fix an algorithm A. For any $t, 0 \leq t \leq \log p - 1$, there is a partial assignment α with at least $\lfloor (p-1)/2^t \rfloor$ free processors with the following property. On any input $\boldsymbol{x}$ consistent with α, the input x_i of a free processor i will be known only to a set of contiguous processors S_i containing i at time t, where $0 \notin S_i$. Moreover, for any two distinct free processors i and j, $S_i \cap S_j = 0$.*

Proof: The proof is by induction on t. The base case is when $t = 0$. In this case, since no communication has taken place, it is immediate that if α is the partial assignment which is not fixed anywhere, then all possible inputs $\boldsymbol{x}$ are consistent with α, all processors in the range $1, \ldots, p-1$ are free, and the value x_i of every processor i is known only to processors in the set $S_i = \{i\}$.

Suppose the lemma is true for $t - 1 \geq 0$, and let α be the partial assignment as in the statement of the lemma. Suppose that at round t, active processors broadcast to the right (the other case, when active processors broadcast to the left, is handled similarly).

We will define a partial assignment α' which extends α and satisfies the lemma for time t. To do this, we consider the free processors at time $t-1$ in order from that with the largest index to that with the smallest index. (We consider processors in the opposite order in the case that the broadcast is to the left.) If free processors j, i occur consecutively in this ordering, with $j > i$, we say that j is i's *free neighbor to the right* at time $t-1$.

For each of these processors i in turn, we will determine whether i remains free at round t, and if not, we will extend α to fix input value x_i. If i does remain free, we will define a corresponding set S_i' containing i, and will show that at time t, on any input $\boldsymbol{x}$ consistent with α', x_i is known only to processors in S_i', that $0 \notin S_i'$ and that $S_i' \cap S_j'$ are disjoint, for free processors $i \neq j$.

Hence consider some processor i that is free at time $t-1$. We say that S_i is *potentially active* if there is some input consistent with α such that some processor in S_i broadcasts at round t with that input. Otherwise S_i is said to be *inactive*.

If S_i is potentially active, then we define i to be free at round t if and only if the following conditions hold: (i) it is not the largest numbered free processor at time $t-1$, and (ii) processor i's free neighbor to the right at time $t-1$ is not free at time t. (Note that since i's free neighbor to the right has index $j > i$, and since we consider the free processors in order from the largest to the smallest, it is already determined whether j is free at time t.) The corresponding set S_i' is defined to be the smallest contiguous set containing S_i and S_j, where j is i's free neighbor to the right at time $t-1$. Otherwise, i is not free at time t and the value of x_i is fixed in α', to force some processor in S_i to be active at round t.

It is important to note that since the processors in S_i do not know the values of x_j for the free processors $j \neq i$ at time $t-1$, then some assignment to the input x_i will force some processor in S_i to be active at round t regardless of any assignment to other inputs whose processors are free at time $t-1$.

If S_i inactive, then we define i to be free at round t and the corresponding set of processors S_i' is to equal S_i. This completes the description of α' and the set S_i' for each free processor i.

We now argue that α' satisfies the lemma at time t. It is straightforward to see from the construction that for each free processor i at time t, $i \in S_i'$ and that S_i' is a set of contiguous processors. Also, for any two distinct free processors, i and j, $S_i' \cap S_j' = 0$. This is because the S_i are contiguous, non-overlapping sets, and each S_i' is either S_i or is formed by "collapsing" two neighboring sets S_i and S_j, where processor j is free at time $t-1$ but not at time t. Finally, using the fact that no set S_i contains processor 0, we show that no set S_i' contains processor 0. This is easy to see if S_i' equals S_i, since we know S_i does not contain processor 0. Otherwise, S_i' is the smallest contiguous set containing S_i and S_j, where j is i's free neighbor to the right at time $t-1$. Since $0 < i < j$, processor 0 cannot lie between the contiguous sets S_i and S_j. This, together with the fact that neither S_i nor S_j contain processor 0, implies that S_i' does not contain processor 0.

We next show that for any $\boldsymbol{x}$ consistent with α', if i is free at time t then x_i is known only to those processors in S_i'. First note that since processor 0 is not in S_i for any processor i that is free at time $t-1$, the instruction broadcast by 0 at time t does not reveal any information about the values of processors which are free at time $t-1$. Also, it is clear that if S_i is inactive at time t, for all $\boldsymbol{x}$ consistent with α', then x_i is still known only to those processors in S_i at time t.

Consider the other case, where S_i is potentially active at time t. Then i's free neighbor to the right at time $t-1$, say processor j, is free at time $t-1$ but not at time t. Moreover, by our construction of α', on any $\boldsymbol{x}$ consistent with α' there is a processor b in S_j which broadcasts at time t. Hence, on any input $\boldsymbol{x}$ consistent with α', any broadcast of a processor in set S_i reaches only processors in the segment between this active processor and processor b. The processors in this segment are contained in the smallest contiguous set containing both S_i and S_j. Hence, x_i is known only to processors in S_i' at time t.

To complete the proof, it remains to show that there are $\geq \lfloor (p-1)/2^t \rfloor$ free processors at time t. By the inductive hypothesis, there are $\geq \lfloor (p-1)/2^{t-1} \rfloor$ free processors at time $t-1$. If i and j are two neighboring free processors at time $t-1$, then at least one of these is still free at time t. To see this, suppose that $i < j$ and that j is not free at time t. Then either S_i is inactive at time t in which case i is free at time t, or S_i is potentially active, in which case both conditions (i) and (ii) are satisfied, so again i is free at time t. Hence the number of free processors at time t is at least $\lfloor \lfloor (p-1)/2^{t-1} \rfloor /2 \rfloor = \lfloor (p-1)/2^t \rfloor$ as required. ∎

The proof of theorem 3.2 now follows easily from Lemma 3.1. If $p \geq 2$ then $\lfloor (p-1)/2^{\log p - 1} \rfloor \geq 1$. Hence by the lemma, there is a partial assignment α which is not fixed at one free processor, say i, with the following property. On any input $\boldsymbol{x}$ consistent with α,

at time $\log p - 1$, the input x_i will be known only to a set of processors S_i, where $0 \notin S_i$. Hence, x_i is unknown to processor 0 at time $\log p - 1$.

Lower bounds of $\log p$ time for PARITY and MINIMUM follow immediately from theorem 3.2, as discussed after the statement of that theorem. The same lower bound must also hold for SUM since PARITY can be computed from SUM without any communication. Thus, we have:

Theorem 3.3 *On a one-dimensional sub-bus mesh with p processors, the time to compute PARITY, SUM, or MINIMUM is at least* $\log p$.

In order to obtain lower bounds for MAJORITY and many other symmetric Boolean functions we need to modify lemma 3.1. If α is a partial assignment define α_0 to be the number of inputs fixed to 0 in α and α_1 to be the number of inputs fixed to 1 in α. We say a partial assignment α is *b-balanced* if $0 \leq \alpha_b - \alpha_{1-b} \leq 1$. That is, α is 1-balanced if the number of inputs assigned to 1 in α is equal to or one greater than the number of inputs assigned to 0 in α. Similarly, α is 0-balanced if the number of inputs assigned to 0 in α is equal to or one greater than the number of inputs assigned to 1 in α.

Lemma 3.2 *Fix an algorithm A. For any bit b and for any $t, 0 \leq t \leq \log_3 p - 1$, there is a b-balanced partial assignment α with at least $\lfloor (p-1)/3^t \rfloor$ free processors with the following property. On any input $\boldsymbol{x}$ consistent with α, the input x_i of a free processor i will be known only to a set of contiguous processors S_i containing i at time t, where $0 \notin S_i$. Moreover, for any two distinct free processors i and j, $S_i \cap S_j = 0$.*

Proof: This proof is similar to that of lemma 3.1. Assume we have a b-balanced partial assignment α at time $t-1$ and a number n of free processors with their associated segments satisfying the condition of the lemma. Assume also, that at time t there is a broadcast to the right. As in the proof of lemma 3.1 a segment is inactive if no processor in the segment would become active on any input consistent with α and is potentially active otherwise. As before any processor i which is free at time $t-1$ and whose segment S_i is inactive remains free at time t. Assume the free processors at time $t-1$ are indexed by $i_1, i_2, ..., i_n$ where $i_j > i_{j+1}$ for $1 \leq j \leq n$. We consider these processors three at a time, largest index to smallest, to determine which potentially active processors remain free at time t and for those that do not remain free, what their inputs will be assigned to in the new b-balanced partial assignment α'. If n is divisible by 3 then this process will end simply. If not, there will be a remaining group of 1 or 2 which must be dealt with.

Assume that for $j \leq 3m$, it has already been determined whether i_j is free at time t and if not, what the assignment to x_{i_j} is in the assignment α'. We now consider the processors i_{3m+1}, i_{3m+2} and i_{3m+3} where $3m+3 \leq n$. There are four cases to consider depending on how many of the segments $S_{i_{3m+1}}, S_{i_{3m+2}}, S_{i_{3m+3}}$ are potentially active. If none are potentially active then there is nothing to do. If exactly one, say $S_{i_{3m+k}}$, is potentially active then set $x_{i_{3m+k}}$ to a value to make α' b-balanced. That is, if $\alpha'_0 = \alpha'_1$ then assign $x_{i_{3m+k}}$ to b, otherwise set it $1-b$. If exactly two of the segments are potentially active, then assign the

input associated with one to 0 and the other to 1. Finally, if all three are potentially active then i_{3m+3} remains free, $x_{i_{3m+2}}$ is set so as force a processor in the segment $S_{i_{3m+2}}$ to be active, and then $x_{i_{3m+1}}$ is set to make α' b-balanced.

Once the groups of three have been processed there may be one or two remaining free processors. If exactly one of the segments in this remaining group is potentially active, then assign the input of that processor to a value to make the partial assignment b-balanced. If exactly two of the segments of the free processors in this remaining group are potentially active, then assign the two inputs of the free processors to opposite values.

At the end of this process, at least $\lfloor n/3 \rfloor$ of the processors are free. If i is free at time t and S_i is inactive, then $S_i' = S_i$. If i is free at time t and S_i is potentially active, then the corresponding set S_i' is defined to be the smallest contiguous set containing S_i and S_j, where j is i's free neighbor to the right at time $t-1$. This happens in the fourth case above when $i = i_{3m+3}$ and $j = i_{3m+2}$.

It should be clear that α' is b-balanced partial assignment with at least $\lfloor (p-1)/3^t \rfloor$ unassigned inputs. Furthermore, for the same reasons as in the proof of lemma 3.1, for any input $\boldsymbol{x}$ consistent with α', the input x_i of a free processor i will be known only to a set of contiguous processors S_i containing i at time t, where $0 \notin S_i$. Clearly, the segments at time t are disjoint. ∎

As a consequence of lemma 3.2 we have the following theorem which allows us to find a b-balanced input with a component unknown to processor 0 at a time slightly less than the maximum time to find just some input with a component unknown to processor 0 as in theorem 3.2.

Theorem 3.4 *On a one-dimensional sub-bus mesh with p processors and for any algorithm A and bit b, there exists a b-balanced input $\boldsymbol{x}$ such that for some $i, 1 \leq i < p-1$, x_i is set to b, but is unknown to processor 0 at time $\log_3 p - 1$.*

Proof: If $p \geq 2$ then $\lfloor (p-1)/3^{\log_3 p - 1} \rfloor \geq 1$. By lemma 3.2, there is a $(1-b)$-balanced partial assignment α which is not fixed at a free processor i. Set $x_i = b$, then set the remaining unassigned inputs so as to make the input b-balanced. Since 0 is not a member of the segment S_i at time $\log_3 p - 1$, then x_i is not known to processor 0. ∎

Theorem 3.5 *On a one-dimensional sub-bus mesh with p processors the time to compute MAJORITY is at least $\log_3 p$.*

Proof: Let A be any algorithm for MAJORITY. There are two cases to consider depending on whether p is even or odd. If p is even then by theorem 3.4 select a 0-balanced input $\boldsymbol{x}$ and an i such that $x_i = 0$ and x_i is not known to processor 0 at time $\log_3 p - 1$. Clearly, processor 0 cannot have computed the majority of the inputs by time $\log_3 p - 1$ since its computation would be identical for the input $\boldsymbol{x}$ and $\boldsymbol{x}'$ which is identical to input $\boldsymbol{x}$ except that $x_i' = 1$. The latter input has a majority of 1's while the former does not. If p is odd then select a 1-balanced input $\boldsymbol{x}$ and an i such that $x_i = 1$ and x_i is not known to processor 0 at time $\log_3 p - 1$. The remainder of the argument is similar to that above. ∎

For any symmetric Boolean function f on p inputs define $m(f)$ to be the k such that $\frac{p}{2} - k \geq 0$ is minimal and for some bit b the value of f on an input with exactly k inputs equal to b differs from the value of f on an input with $k+1$ inputs equal to b. For example, $m(\text{MAJORITY}) = m(\text{PARITY}) = \lfloor p/2 \rfloor$ and $m(\text{OR}) = m(\text{AND}) = 0$.

Corollary 3.1 *On a one-dimensional sub-bus mesh with p processors the time to compute any symmetric function f is at least* $\log_3(2m(f))$.

Proof: Let f be given. Let b be such that if $m(f)$ inputs have the value b then f has one value and if $m(f)+1$ inputs have the value b then f has another value. For simplicity consider the case in which $m(f)$ inputs equal 0 implies the value of f is 0 and $m(f)+1$ inputs equal 0 implies the value of f is 1. If exactly $p - 2m(f)$ inputs are set to 0 then the restricted function has $2m(f)$ inputs. By a proof identical to the proof of theorem 3.5 any algorithm to compute the restricted function must take time $\log_3(2m(f))$. The argument for $b = 1$ is similar. ∎

Define THRESHOLD_k to be the Boolean function which is 0 with k or fewer inputs set to 1 and 1 otherwise. Clearly, $m(\text{THRESHOLD}_k) = \min(k, p-k)$. Thus, we have the following:

Corollary 3.2 *On a one-dimensional sub-bus mesh with p processors the time to compute $THRESHOLD_k$ is at least* $\log_3(2\min(k, p-k))$.

4 Algorithms for PARITY and SUM

In this section we present asymptotically optimal algorithms for PARITY and SUM on the two-dimensional sub-bus mesh computer. We start with the PARITY algorithm. It is the simpler of the two, and introduces some of the key ideas which are useful in the SUM algorithm.

4.1 PARITY Algorithm

We will introduce a series of problems, in increasing order of difficulty. The algorithm for each problem will lead to the next one with some fresh tricks. This will help us concentrate on one idea at a time.

Each of the algorithms below can be executed on a sub-mesh of the $\sqrt{p} \times \sqrt{p}$ mesh. By an *array* or *sub-array* we mean a sub-mesh of the full mesh which may be non-square and non-contiguous. In the case it is non-contiguous it is assumed that the processors between any two processors in the sub-array are inactive so as not to interfere with communication between the processors in the sub-array. Furthermore, any of the algorithms below can be executed in parallel on disjoint and properly aligned sub-arrays of the full $\sqrt{p} \times \sqrt{p}$ array. If the algorithm is executed on a $m \times n$ sub-array, then we say processor (i, j) is the processor in the (i, j)-th position (the i-th column and j-th row) of the sub-array where $0 \leq i < m$ and $0 \leq j < n$. Although it is not generally the case that processor (i, j) has its `PIDx` $= i$

and `PIDy` $= j$, it will always be the case that i, j, and dimensions of the sub-array can be computed from the `PID` of the processor and other local data in constant time.

Lemma 4.1 *On an $n \times 2^n$ array with each processor in the top row having an input bit, the parity of the input bits can be computed in constant time.*

Proof: There are 2^n possible inputs, so we will make row j of the array responsible for determining whether the input, thought of as an integer x with $0 \leq x < 2^n - 1$, actually equals j. In particular processor (i, j) determines if the input in processor $(i, 0)$ equals the i-th bit of j. A downward broadcast of the input gives processor (i, j) knowledge of the input in processor $(i, 0)$. Then processor (i, j) compares the input of processor $(i, 0)$ with the i-th bit of j. A constant time AND of the outcomes of these comparisons in all the rows in parallel, tells processor $(0, j)$ whether the input, thought of as an n bit number, equals j. This information can then be broadcast up to processor $(0, 0)$. Since $2^n \leq \sqrt{p}$, we know $j \leq \log p$ so that processor $(0, 0)$ can compute the sum of the bits in j in constant time, using the fact that omputing the sum of the bits of an input is in uniform NC. The parity of the input bits is the parity of this sum. ∎

We saw that with exponentially many rows we can compute the parity in constant time. In general, if we have more than a constant number of rows, we can beat the straightforward $O(\log n)$ time algorithm.

Lemma 4.2 *On an $n \times m$ array with each processor in the top row having an input bit, the parity of the input bits can be computed in time $O(\frac{\log n}{\log \log m})$.*

Proof: We can think of the $n \times m$ array as $n/\log m$ sub-arrays of dimension $\log m \times m$ placed side by side. As in the previous proof, we can compute the parity of groups of $\log m$ bits in constant time. This leaves $n/\log m$ partial results in the first row of an array of dimension $\frac{n}{\log m} \times m$. Repeating the process $\frac{\log n}{\log \log m}$ times we have the parity of all the n bits. ∎

So far we have been assuming that only the processors in the top row have inputs. Let us now consider the case where each processor has an input.

Lemma 4.3 *On an $n \times m$ array with each processor having an input bit, the parity of the input bits can be computed in time $O(\log m + \frac{\log n}{\log \log m})$.*

Proof: First, in parallel, the processors within each column run the one-dimensional PARITY algorithm described in section 3.1. This part takes time $O(\log m)$. At this point, we have partial results stored in the top row. From the previous lemma, the parity of these partial results can be computed in an additional $O(\frac{\log n}{\log \log m})$ steps. ∎

We are ready to give our PARITY algorithm.

Theorem 4.1 *On a $\sqrt{p} \times \sqrt{p}$ mesh with each processor having an input bit, PARITY can be computed in time $O(\frac{\log p}{\log \log p})$.*

Proof: Think of the $\sqrt{p} \times \sqrt{p}$ mesh as $\sqrt{p}/m$ smaller arrays of dimension $\sqrt{p} \times m$ one on top of the other. Each of these arrays computes the parity of their input bits in parallel. By previous lemma, this takes $O(\log m + \frac{\log \sqrt{p}}{\log \log m})$ time and leaves $\sqrt{p}/m$ partial results in the leftmost column. By lemma 4.2 their parity can be computed in time $O(\frac{\log \sqrt{p}}{\log \log (\sqrt{p}/m)})$. Choosing $\log m = \frac{\log p}{\log \log p}$ we get a total running time of $O(\frac{\log p}{\log \log p})$. ∎

4.2 SUM Algorithm

Computing PARITY is the same as computing the sum of the inputs modulo 2. Lemmas 4.1 and 4.2 can be generalized to compute the sum, modulo a small integer, of inputs on the top row. For all the problems below we assume that the inputs are non-negative integers of length $O(\log p)$.

Lemma 4.4 *If $\log Q \leq \sqrt{\log n}$ then on an $n \times n$ array with each processor having Q and with each processor in the top row having an input integer, the sum of the inputs modulo Q can be computed in time $O(\frac{\log n}{\log \log n})$.*

Proof: Let $m = \frac{\log n}{\log Q}$. Let us focus on an $m \times n$ sub-array which has m inputs on the top row. There are $Q^m = n$ possibilities for the m inputs mod Q. For $0 \leq j < n$, think of j as an integer written in base Q. As in the computation of parity, processor (i, j) is responsible for determining if the i-th input mod Q is equal to the i-th Q-ary digit of j. Processor (i, j) learns of the input at $(i, 0)$ by a broadcast down from the first row. Then, processor $(0, j)$ will learn from an AND on its row that the i-th Q-ary digit of j is the i-th input mod Q for all i such that $0 \leq i < m$. Processor $(0, j)$ then transmits j to processor $(0, 0)$ where the sum mod Q of the Q-ary digits of j is computed.

The original $n \times n$ array can be divided into n/m sub-arrays of dimension $m \times n$ placed side by side where the algorithm above is performed in parallel. What remains are n/m numbers in the top row. Iterate this process $O(\frac{\log n}{\log m}) = O(\frac{\log n}{\log \log n - \log \log Q})$ times to compute the sum of all the n integers modulo Q.

To complete the proof we must argue that computing the sum mod Q of the Q-ary digits of a number x is in uniform NC. We assume both Q and x are written in binary. First, since $\log Q \leq \sqrt{\log n}$ and $n \leq \sqrt{p}$, then the length of Q is $O(\sqrt{\log p})$. Second, $x \leq \sqrt{p}$, so that x is of length $O(\sqrt{\log p})$. Thus, the lengths of Q and x can be assumed to be bounded by the same number b, which we can assume is a power of two and of lenght $O(\sqrt{\log p})$. To find the sum mod Q of the Q-ary digits of x write x as $x_0 + Q^{b/2} x_1$, by dividing x by $Q^{b/2}$. Recursively, find the a_0 and a_1 which are the sums mod Q of the Q-ary digits of x_0 and x_1 respectively. Then, $a = (a_0 + a_1) \bmod Q$ is the sum mod Q of the Q-ary digits of x. The necessary powers of Q, division by these powers, and sum are all in uniform NC. Since the number of levels of recursion is bounded by $\log_2 b$, then the result a can also be computed in uniform NC. ∎

By the Chinese remainder theorem we know that if we can compute the sum modulo sufficiently many small integers, we can compute the exact sum.

Lemma 4.5 *If $6 \leq \log t \leq \sqrt{\log n}$ then on an $n \times tn$ array with each processor in the top row having an input integer such that the sum of these integers is less than 2^t, the sum of the inputs can be computed in time $O(\frac{\log n}{\log\log n})$.*

Proof: Think of the $n \times tn$ array as t sub-arrays, each of size $n \times n$, one on top of the other. Let M be the product of all primes less than t. If $t \geq 41$ then $M \geq 2^t$ [27, Corollary to Theorem 4, page 70]. Hence, if the sum of the input integers is less than 2^t then it is enough to compute the sum modulo M. We already know how to compute the sum modulo small primes. Our plan is to let each $n \times n$ sub-array compute the sum modulo a different prime and then apply the Chinese remainder algorithm to compute the sum modulo M.

To begin with the processors in the top row broadcast the input values down the columns. The jth sub-array decides whether j is a prime. This can be done in two stages. A number j is prime if and only if it is not divisible by any number between 1 and $\sqrt{j}$. In the first stage, assign $\sqrt{j}$ processors in the first row to check for each possible divisor. In the second stage, these processors compute an AND of their results. Only processors in the j-th sub-array for prime j participate in all subsequent steps. The j-th sub-array computes a_j, the sum of all the inputs modulo j. By Lemma 4.4, this can be done in $O(\frac{\log n}{\log\log n})$ time.

Next, in $O(\log t)$ steps, each processor in the j-th sub-array computes M, the product of all primes less than t, and $m_j = M/j$. The processors in the j-th sub-array compute $(a_j m_j)((m_j)^{-1} \bmod j)$. This can be done in constant time. The nontrivial part is computing $((m_j)^{-1} \bmod j)$. There are at most j possible values for the inverse. We assign j processors in (say) the first row of the j-th sub-array for each possible value of the inverse. In one step, each of these assigned processors can check whether it has the right value of inverse. The processor corresponding to the right value of the inverse broadcasts this to all other processors. By Chinese remainder theorem, the exact value of sum is summation of $(a_j m_j)((m_j)^{-1} \bmod j)$, which can be computed in $O(\log t)$ steps.

In total the computation can be done in $O(\log t + \frac{\log n}{\log\log n}) = O(\frac{\log n}{\log\log n})$ time. ∎

Lemma 4.6 *If $tw \leq m$ and $6 \leq \log t \leq \sqrt{\log w}$ then on an $n \times m$ array, with each processor in the top row having an input integer such that the sum of these integers is less than 2^t, the sum of the inputs can be computed in time $O(\frac{\log n}{\log\log w})$.*

Proof: Think of the $n \times m$ array as n/w subarrays each of dimension $w \times m$ side by side. Since $tw \leq m$, each $w \times m$ subarray can be thought of as containing a $w \times tw$ subarray at the top. Each $w \times tw$ subarray has its input on the top row. By lemma 4.5, the sum of the inputs of all the $w \times tw$ subarrays can be computed in time $O(\frac{\log w}{\log\log w})$ leaving n/w partial sums in the top row. This process is repeated $\log n / \log w$ times until the input is reduced to a single number. This reduction takes time $O(\frac{\log n}{\log\log w})$. ∎

We now consider the case where *each* processor has an input.

Lemma 4.7 *If $tw \leq m$ and $6 \leq \log t \leq \sqrt{\log w}$ then on an $n \times m$ array with each processor having an input integer such that the sum of these integers is less than 2^t, the sum can be computed in time $O(\log m + \frac{\log n}{\log\log w})$.*

Proof: The first step is simply to add the columns in parallel in $O(\log m)$. We are now reduced to the problem in the previous lemma. Hence the total time is $O(\log m + \frac{\log n}{\log\log w})$. ∎

Theorem 4.2 *On a $\sqrt{p} \times \sqrt{p}$ mesh with each processor having an input integer of length $O(\log p)$, SUM can be computed in time $O(\frac{\log p}{\log\log p})$.*

Proof: Choose $t = \log(p^k)$ where p^k is an upper bound on the sum of the input integers and $\log t \geq 6$. Let c be a constant, which depends only on k, such that $\log t \leq \sqrt{c \log p / \log\log p}$. Now, choose w such that $\log w = c \log p / \log\log p$. Let $m = tw$, so that t, w, and m satisfy the hypothesis of lemma 4.7. Think of the $\sqrt{p} \times \sqrt{p}$ mesh as $\sqrt{p}/m$ arrays of dimension $\sqrt{p} \times m$ one on top of the other. By lemma 4.7 the sum of these arrays can be computed in time $O(\log m + \frac{\log \sqrt{p}}{\log\log w})$, leaving $\sqrt{p}/m$ partial sums in the first column. We may now apply lemma 4.6 to these inputs to find the full sum in an additional $O(\frac{\log \sqrt{p}}{\log\log w})$ time. The total time is then $O(\log m + \frac{\log \sqrt{p}}{\log\log w})$. Since $\log w = c \log p / \log\log p$ and $\log m = \log w + \log t = O(\frac{\log p}{\log\log p})$, the total time of the algorithm is $O(\frac{\log p}{\log\log p})$. ∎

It is interesting to note, that if we assume that the individual processors can operate on integers of arbitrary length in constant time, then using the technique of theorem 4.2, the sum of p integers of length $2^{O(\sqrt{\log p / \log\log p})}$ can be computed in time $O(\frac{\log p}{\log\log p})$.

Since MAJORITY can be computed from SUM in constant time we may now state the theorem:

Theorem 4.3 *On a two-dimensional sub-bus mesh with p processors, PARITY, MAJORITY, and SUM can be computed in time $O(\frac{\log p}{\log\log p})$.*

In the next section we will show these bounds are optimal.

5 Simulation of a Sub-Bus Mesh by a CRCW PRAM

In this section, we prove a $\Omega(\frac{\log p}{\log\log p})$ lower bound for computing PARITY, MAJORITY, and SUM on the two-dimensional sub-bus mesh computer. To prove this, we show that any algorithm for the sub-bus model can be simulated by a CRCW (concurrent read, concurrent write) PRAM algorithm with only a constant factor loss in running time. We then apply lower bound results for PARITY on the PRAM model. We begin this section by describing the PRAM results. Then we describe our lower bound model and describe the simulation in detail.

Beame and Hastad [4] considered lower bounds for the following "ideal" CRCW PRAM model. There are $p(n)$ numbered processors which share $c(n)$ numbered memory cells, where $p(n)$ and $c(n)$ are polynomially bounded. There is no bound on the possible contents of a memory cell. Initially, the input bits $x_0, \ldots, x_{n-1}$ are stored in the first n memory cells and the remaining cells have value 0. Before each step t, a processor is in some state, say q.

At the tth step, the processor may read the value v stored in some memory cell C. Based on C, v and q the processor moves to a new state q', and may write a value v' to some cell C'. There is no limit on the number of states of a processor nor on the resources needed to compute v' and C'. If several processors attempt to write into the same memory cell at the same step, the lowest numbered processor succeeds. This model is called the *priority* CRCW PRAM. Beame and Hastad have shown that the time to compute any of PARITY, MAJORITY, and SUM on the ideal CRCW PRAM is $\Omega(\log n / \log\log n)$ [4].

In our lower bound model of the two-dimensional sub-bus there are p processors, numbered $0, 1, \ldots, p-1$, connected in a $\sqrt{p} \times \sqrt{p}$ mesh as in the two-dimensional upper bound model. Processor 0 is the front-end. The computation proceeds in rounds costing one unit, and in each round, the processors first communicate and then perform arbitrary internal computation. The communication is done just as in the upper bound model. Again, there is no bound on the size of values broadcast or computed. The main result of this section is the following.

Theorem 5.1 *Any problem which can be solved in time $t(p)$ on a two-dimensional sub-bus mesh computer with p processors can be solved in time $O(t(p))$ on a priority CRCW PRAM with $O(p^{3/2})$ processors.*

Proof: Let A be a $T(p)$-time algorithm for solving a problem on the two-dimensional sub-bus lower bound model. We describe an ideal CRCW PRAM that simulates A, such that each round of A takes $O(1)$ steps.

In the simulating PRAM, there are p processors, numbered $0, 1, \ldots, p-1$, corresponding to the p processors of the sub-bus model. There are also auxiliary processors, whose computation will be described later.

There are two special memory cells called Condition and Statement, used by processor 0 to communicate the instruction at each round. Also, corresponding to each processor $i, 0 \leq i \leq p-1$, there are the following special memory cells (we do not specify their exact addresses, but assume they are computable by processor i). Active(i) is used to denote at each round whether i is active. It is initialized to false at the beginning of each round. Send(i) is used to store the value broadcast by i, at each round, if i is active. Receive(i) is used to store the value, if any, received by i in each round. At the start of each round, processor i sets Active(i) to false and sets Receive(i) to some special value which is not in the range of possible values that can be broadcast by A.

We now describe the simulation of a single round of A. First, processor 0 writes the strings <*condition*> and <*statement*> in cells Condition and Statement, respectively. Each processor $i, 0 \leq i \leq p-1$ reads these cells and decides if it is active at this round. If so, i writes the value to be broadcast in Send(i) and sets Active(i) to true.

We next describe how each processor i determines the value it receives (if any) during the broadcast instruction. If i receives a value, it is from one of $\sqrt{p}$ processors on either the vertical or horizontal bus along which i is connected. Let these processors be numbered $i_1, \ldots, i_{\sqrt{p}}$, where the ordering is such that if i_1 is active, then i_1 is the processor from which i receives a message; if i_1 is not active but i_2 is then i_2 is the processor from which i receives

a message, and so on, so that if i_k is active and none of $i_1, \ldots, i_{k-1}$ are active, then i_k is the processor from which i receives a message. For example, if the direction of communication is up, then the sequence is $(i + \sqrt{p}) \bmod p, (i + 2\sqrt{p}) \bmod p, \ldots, (i + \sqrt{p}\sqrt{p}) \bmod p$.

Each processor i has $\sqrt{p}$ auxiliary processors to help it compute the value it receives, if any. Let the auxiliary processors of i be numbered $n_i+1, \ldots, n_i+\sqrt{p}$, where $n_i = p+i\sqrt{p}-1$. Each processor $n_i + k$ computes i_k; this can be done by reading $<$*statement*$>$, to determine the direction of communication. If processor i_k is active, that is, Active(i_k) is true, then processor n_i+k reads the value Send(i_k) and writes it in Receive(i). Because of the ordering of the auxiliary processors, and the priority write conflict resolution assumption, the value written in Receive(i) is the value received by processor i at that round of A, in the sub-bus computation.

Once the cells Receive(i) have been computed, each processor $i, 0 \leq i \leq p-1$ completes the round by simulating the internal computation of the ith sub-bus processor. This completes the description of the simulation. It is clear that all the steps described can be done by the processors of the ideal CRCW PRAM, since they have unbounded resources with which to compute at each step, and an unbounded number of states which can be used to store the internal configurations of the processors of the sub-bus mesh computer. ∎

As a direct consequence of theorem 5.1 we have:

Theorem 5.2 *On a two-dimensional sub-bus mesh with p processors the time to compute PARITY, MAJORITY, and SUM is* $\Omega(\frac{\log p}{\log\log p})$.

6 MINIMUM

In this section we survey the two-dimensional complexity of MINIMUM in the comparison model of the sub-bus mesh. Previous work for the reconfigurable mesh shows that the time to compute MINIMUM on the two-dimensional mesh is $\Theta(\log\log p)$. For completeness, we include the algorithm, adapted to the sub-bus model.

Theorem 6.1 *[21, 31] In the comparison model of a two-dimensional sub-bus mesh computer with p processors the time to compute MINIMUM is* $\Theta(\log\log p)$.

Proof: Recall, that in the comparison model we assume the only operations allowed on input values are comparison, copy, and broadcast. With this restriction any sub-bus mesh algorithm for MINIMUM can be thought of as a "parallel comparison tree" as defined by Valiant [31]. In this model, any p comparisons of arbitrary input values can be made in one step. Depending on the outcome of these comparisons one of 2^p branches to the next step can be made. Valiant showed that in the parallel comparison tree model with p processors, $\Omega(\log\log p)$ steps are necessary to determine the minimum of p inputs.

In the same paper [31], Valiant gave an algorithm for the minimum which runs in $O(\log\log p)$ time. The Valiant algorithm can be realized on the reconfigurable mesh computer as shown by Miller *et al* [21]. In fact, their work also applies to the two-dimensional sub-bus mesh computer. In the Valiant algorithm, assume there are p processors where

$p = 2^{2^k}$ for some k. With one parallel comparison the number of possible minima to consider is reduced to $p/2$. Subsequently, if there are p/b possible minima remaining then in one parallel comparison this number can be reduced to p/b^2. This is done by dividing the p/b numbers into p/b^2 groups of size b and using one parallel comparison to find the minimum of all the groups simultaneously. A group of size b requires b^2 processors to find the minimum in one parallel comparison. Thus, p processors are utilized to find the p/b^2 possible minima.

The Valiant algorithm can be realized on the two-dimensional sub-bus mesh computer. The basic building block is the MINIMUM algorithm described in section 2.2, which in constant time finds the minimum of n values if they are in the first row of an $n \times n$ array of processors. In the two-dimensional algorithm, if there are p/b values remaining (for $b \geq 2$) then there are p/b^2 sub-arrays of the $\sqrt{p} \times \sqrt{p}$ array, each which is $b \times b$ with b inputs in the first row of the sub-array. The left-most corner of the sub-arrays appear at processors (ib, jb) where $0 \leq i, j < \sqrt{p}/b$. Using the basic building block, the p/b possible minima can be reduced to p/b^2 possible minima in constant time. These possible minima are located at processors indexed (ib, jb) where $0 \leq i, j < \sqrt{p}/b$. By first broadcasting these values to the right, then selectively broadcasting these values up, we end up with p/b^2 remaining values in the first row of p/b^2 sub-arrays each of size $p/b^2 \times p/b^2$. The iterative version of this algorithm is given below:

MINIMUM of p values on a two-dimensional mesh
input: plural number variable `x`
output: MINIMUM value $\mathtt{x}_{i,j}$
other: plural number variable `y`, boolean variable `t`, singular integer `b`
NOTE: We assume $p = 2^{2^k}$ for some k.

```
begin
   b ← 2
   broadcast_up[1].y ← x
   if y < x then
      x ← y
   repeat begin
      Step 1
      if PIDy mod b = 0 then
         broadcast_down[b-1].x ← x
      if PIDx mod b = PIDy mod b then begin
         y ← x
         broadcast_left[b-1].y ← y
      endif
      if PIDx mod b = 0 then
         broadcast_right[b-1].y ← y
      t ← (x > y)
      if t then
         broadcast_up [b-1].t ← true
```

```
      if not t then
        broadcast_left[b-1].x ← x
      Step 2
      if b = p then
        return x_{0,0}
      if PIDy mod b = 0 then begin
        if PIDx mod b = 0 then
          broadcast_right[b-1].x ← x
        if PIDx mod b = (PIDy mod b^2) / b then
          broadcast_up[b^2-1].x ← x
      endif
      b ← b^2
    endrepeat
end
```

To explain the algorithm in more detail, at the beginning of step 1, there are possible minima in every processor along every b-th row. Then step 1 does the constant time MINIMUM on each $b \times b$ block in parallel. Within each block, it distributes the b initial values into the variables `x` and `y` for each processor. Thus, the processor in position (i, j) in each block has the initial value of processor $(i, 0)$ in its `x` register and the initial value of processor $(j, 0)$ in its `y` register. A comparison of `x` and `y` is recorded in the boolean value `t`. After broadcasting `t` up the only processor with the value `false` is the processor in position $(i, 0)$ which has the minimum for this block. So, that processor will broadcast its initial value `x` to processor $(0, 0)$ within the block.

In step 2, we have the situation where the processors (ib, jb) have possible minima, and we want to move them all to rows, so that the processors (i, jb^2) all have potential minima. This is accomplished in two sub-steps. First, broadcast the potential minima to the right. Second, selectively broadcast the minima up. That is, each potential minima at processor $(ib + k, jb^2 + kb)$ for $0 \leq k < b$ is broadcast up.

In case p is not of the form 2^{2^k} for some k then the algorithm must be modified slightly. In the modified algorithm we will always maintain an active rectangular sub-mesh which is $\sqrt{p} \times q$ where $q \leq \sqrt{p}$ and b divides q evenly. In attempting step 2 in may happen that b^2 does not divide r evenly. If this is the case we set $q' = b^2 \lfloor q/b^2 \rfloor$ and use the upper $\sqrt{p} \times q'$ sub-array. The blocks below this new rectangle can easily be merged into the blocks directly above them in constant time. Thus, in constant time we go from $b \times b$ blocks to $b^2 \times b^2$ blocks. This is sufficient to solve the problem in time $O(\log\log p)$. ∎

7 Conclusions

We have proved tight bounds (to within constant factors) on the time needed to compute several functions on the sub-bus mesh computer. For some of these problems, such as

PARITY, MINIMUM and SUM, the running times on a sub-bus mesh computer match (to within constant factors) the running times on a general PRAM. Moreover, machines based on the sub-bus mesh architecture are commercially available [5]. For these reasons, we believe that the sub-bus mesh architecture deserves further study.

Our algorithms for PARITY and SUM are probably not practical for any reasonable size p for two reasons. First the speed-up by a factor of $O(\log\log p)$ has too large a constant factor to be significant. Second, it is doubtful that hardware designers would want to implement the new NC functions required by the algorithm. It is possible to remove the second factor inhibiting practicality by adding preprocessing phases to the algorithms. A preprocessing phase uses only standard arithmetic/boolean operations to compute values which depend only on the processor PID's and the structure of the algorithm and which in the original algorithm would be computed as results of NC functions. Using preprocessing many more values would be computed than are actually used in the algorithm since during the preprocessing it is not known exactly which values will be needed later on. The necessary preprocessing for the two algorithms PARITY and SUM is complicated, but can be accomplished within the $O(\frac{\log p}{\log\log p})$ time bound.

We have implemented the $O(\log\log p)$ MINIMUM algorithm on a 1,024 processor MasPar MP-1. The constant factor in front of the $\log\log p$ forces the algorithm to run more slowly than the standard $O(\log p)$ algorithm. However, we believe that the ideas in the $O(\log\log p)$ MINIMUM algorithm have the potential to be used in a competitive practical algorithm for finding the minimum on commercially available meshes with more than 1,024 processors.

Several open questions are suggested by our results. Are there simpler $O(\frac{\log p}{\log\log p})$ algorithms for PARITY or SUM on the two-dimensional sub-bus mesh, that may be competitive on real machines? Is it possible to improve the lower bound for MAJORITY on a one-dimensional mesh from $\log_3 p$ to $\log_2 p$? Can our lower bound for PARITY on the two-dimensional sub-bus mesh can be simplified, or improved by a constant factor, using the mesh model directly rather than translating results from the PRAM model?

8 Acknowledgements

We thank Eric Bach, David Barrington, and Paul Beame for several useful discussions and suggestions. We thank Tosten Suel and Phil MacKenzie for providing us with several references to previous work on the reconfigurable mesh. We thank the anonymous referees for useful suggestions and for pointing out a seminal reference to the sub-bus mesh architecture.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. Proceedings of Fifteenth Annual ACM Symposium on Theory of Computing, pp. 1-9, 1983.

[2] A. Bar-Noy and D. Peleg. Square meshes are not always optimal. *IEEE Transactions on Computers*, Vol. 40, pp. 138-147, 1991.

[3] K.E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, Vol. *C*-29, pp.836-840, 1980.

[4] P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *Journal of the ACM*, Vol. 36, pp. 643-670, July 1989.

[5] T. Blank. The MasPar MP-1 architecture. Proceedings of COMPCON Spring 90 - The Thirty-Fifth IEEE Computer Society International Conference, pp. 20-24, February 1990.

[6] G.E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT Press. Cambridge, MA, 1990.

[7] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, vol. 17, pp. 770-785, 1988.

[8] S. A. Cook. A taxonomy of problems with fast parallel algorithms, *Information and Control*, Vol. 64, pp. 2-22, 1985.

[9] S. A. Cook, C. Dwork and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM Journal on Computing*, Vol 15, No. 1, pp. 87-97, February 1986.

[10] F. Fich, P. Radge, and A. Wigderson. Relations between concurrent write models of parallel computation. *SIAM Journal on Computing*, Vol 17, pp. 606-627, 1988.

[11] F. E. Fich, F. Meyer auf der Heide, P. Ragde and A. Wigderson. Lower bounds for parallel random access machines with unbounded shared memory. *Advances in Computing Research*, Vol. 4, pp. 1-15, 1987.

[12] E. Hao, P. D. MacKenzie and Q. F. Stout. Selection on the reconfigurable mesh. *Frontiers of Massively Parallel Computing*, pp. 38-45, 1992.

[13] W.D. Hillis and G.L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, Vol. 29, pp. 1170-1183, December 1986.

[14] J. Ja'Ja'. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[15] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM,* Vol. 27, pp. 831-838, 1980.

[16] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, vol. C-34, pp. 344-354, 1985.

[17] T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

[18] H. Li and Q.F. Stout, Editors. *Reconfigurable Massively Parallel Computers.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[19] P. D. MacKenzie. A separation between reconfigurable mesh models. Proceedings of the 7th International Parallel Processing Symposium, pp. 84-88, 1993.

[20] Y. Matias and A. Schuster. On the power of the 2xn reconfigurable mesh. Manuscript from ATT Bell Labs, June 1993.

[21] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis and Q. F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, Vol. 42, No. 6, pp. 678-692, 1993.

[22] R. Miller and Q. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers,* Vol. 38, pp. 321-340, 1989.

[23] K. Nakano. An efficient algorithm for summing up binary values on a reconfigurable mesh. Research Report N0. 93-003, Advanced Research Laboratory, Hatoyama, Saitama 350-03, Japan.

[24] K. Nakano, T. Masuzawa and N. Tokura. A sub-logarithmic time sorting algorithm on a reconfigurable array. *IEICE Transactions*, Vol. E74, No. 11, pp. 3894-3901, 1991.

[25] V.K. Prasanna Kumar and C.S. Raghavendra. Array processor with multiple broadcasting. *Parallel and Distributed Computing*, Vol. 4, pp. 173-190, 1987.

[26] D. Reisis and V.K. Prasanna Kumar. VLSI arrays with reconfigurable buses. *Supercomputing*, 1st International Conference, Athens, Greece, June 8-12, 1987, Proceedings. Lecture Notes in Computer Science, vol. 297, Springer-Verlag, Berlin, pp. 732-743, 1988.

[27] J.B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, Vol. 6. pp. 64-74, 1962.

[28] W. L. Ruzzo, On uniform circuit complexity, *Journal on Computer and System Sciences*, Vol. 22, pp. 365-383, 1981.

[29] Q.F. Stout. Meshes with multiple buses. 27th Annual IEEE Symposium on Foundations of Computer Science. pp. 264-273, 1986.

[30] C. Thompson. Area-time complexity for VLSI. Proceedings of Eleventh Annual ACM Symposium on Theory of Computing, pp. 81-88, 1979.

[31] L. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, Vol. 4, pp. 348-355, 1975.