An Anomaly in Unsynchronized Pointer Jumping in Distributed Memory Parallel Machine Model

Sun B. Chung
Department of Quantitative Methods and Computer Science
University of St. Thomas
sbchung@stthomas.edu

Abstract

Pointer jumping is one of the most fundamental operations in parallel computing. Pointer jumping on a list of n elements consists of $O(\log_2 n)$ jumping steps. Traditionally, these steps are synchronized. That is, at the end of each jumping step, processors wait until all the processors complete their work for the current step. Theoretically, however, pointer jumping doesn't require synchronization. Without jeopardizing the correctness of execution, a processor may proceed to the next step as soon as it completes the current step.

We implemented both synchronized and unsynchronized pointer jumping on a parallel computer with distributed memory. We expected the unsynchronized implementation to run faster because processors were not forced to idle. To the contrary, we observed an anomaly: the unsynchronized implementation ran more slowly.

We present empirical data and explain how the absence of synchronization causes such an anomaly to occur.

Introduction

Pointer jumping is one of the most fundamental operations in parallel computing. It offers an optimal subroutine for many parallel applications. Pointer jumping on a list of n elements consists of $O(\log_2 n)$ jumping steps. Traditionally, these steps are synchronized. That is, at the end of each jumping step, processors wait until all the processors complete their work for the current step. Theoretically, however, pointer jumping doesn't require synchronization. It means that a processor that has completed its work for a step *could* immediately enter the next step, instead of idling while it waits for other processors to complete the current step.

We implemented both synchronized and unsynchronized pointer jumping on a parallel computer and observed their performance in terms of running time. We expected the unsynchronized implementation to run faster because processors would not be forced to idle. To the contrary, we observed an anomaly: the unsynchronized implementation ran more slowly.

The anomaly was analogous to *Belady's anomaly*, a well-known phenomenon in operating systems: The First-In First-Out page replacement algorithm using four page frames causes, on certain data sets, more page faults than when using three page frames [11].

The primary cause of the anomaly in unsynchronized pointer jumping is the imbalance of workload among processors. For synchronized pointer jumping, the imbalance is not any more serious than for other parallel applications. For unsynchronized pointer jumping, however, the imbalance gets far worse because of the absence of synchronization.

We present empirical data for both synchronized and unsynchronized implementation of pointer jumping. We explain why the absence of synchronization causes huge imbalance among processors as pointer jumping proceeds, whereas the workload among processors is well balanced during synchronized pointer jumping.

Parallel Machine Models, Assumptions, and Related Work

Our target architecture is an asynchronous distributed memory model, in which P processors communicate using messages. One of the most significant characteristics of our model is that the communication cost is much more expensive than computation cost.

The parallel machine that we used for implementing pointer jumping was CM-5, a distributed memory parallel machine with 64 processors. We distributed data randomly and evenly among processors before the execution of pointer jumping began. We experimented with data sets of different sizes using different number of processors. In this paper, we present results that we obtained using a list of 65,536 elements distributed on 32 processors.

Krishnamurthy *et al.* implemented a parallel algorithm for the connected components problem on the CM-5 [8]. Two things are worth mentioning with regard to their work. First, they use the Split-C language, which provides the abstraction of a global address space. Second, they report experimenting synchronous and asynchronous pointer doubling as subroutines for their implementation. However, their use of the term *asynchronous* means more than just the absence of synchronization.

Chung and Condon experimented with different variants of pointer jumping in their parallel implementation of Borůvka's minimum spanning tree algorithm [4].

There are parallel machine models that are different from ours. PRAM (Parallel Random Access Machine) is a theoretical model that assumes *shared memory*. In this model, each processor can access any memory location in one step. The PRAM model "provides an abstraction that strips away problems of synchronization, memory granularity, reliability and communication delays" [5].

Cole and Zajicek proposed a model called APRAM (asynchronous PRAM) in an effort to make explicit the cost of synchronization [5]. Gibbons introduced the Asynchronous PRAM model [6].

Nishimura identifies pointer jumping as one of the "three basic paradigms that serve as building blocks for may parallel algorithms" [10]. She discusses asynchronous pointer jumping in Gibbons's model and states that, in his model, "the insertion of the synchronization barriers between phases ensures that the algorithm is slowed to the speed of the slowest processor each phase." Pointer jumping involves lots of inter-processor communication but, as Nishimura asserts, it "inherently requires no synchronization." These two statements of Nishimura are relevant to the experiments we did, even though the machine models are different.

In the 1990's, many asynchronous parallel algorithms have been designed for various problems, including connected components [7], list ranking and transitive closure [9], and union-find [1].

More recently, Ben-Asher proposed a model called 2PPRAM (two parties PRAM) and presented "a new variant of asynchronous pointer jumping which is work-efficient compared with the common pointer jumping" [2]. The 2PPRAM model is an extension of the PRAM model and assumes shared memory. On our distributed memory model, what he means by "work" roughly corresponds to the "total number of messages sent" during the execution of a parallel algorithm [3].

For the purpose of the paper, *synchronized* and *synchronous* (and similarly *unsynchronized* and *asynchronous*) can be used interchangeably. However, we will use *synchronized* and *unsynchronized* consistently in the rest of the paper.

General Characteristics of Synchronization

A typical characterization of synchronization is that, for many parallel applications, it is *required*. Such applications will not run correctly without synchronization of processors at certain steps. Pointer jumping is not such an application.

A second characteristic of synchronization is that it incurs *overhead*. However, we will ignore this particular overhead, assuming that synchronization among P processors can be done in O (log P) time. Instead, we consider idling of processors during synchronized steps a more serious factor in the slowdown of execution.

Synchronized Pointer Jumping

Let's consider the problem of finding the root of a list: Given a list of n elements, find, for each element i, its root. Each element i has its successor information or successor pointer, $s(i)$. Sequentially there is a simple algorithm that runs in O (n) time, illustrated in Figure 1 at an abstract level: Begin at the tail, follow the successor pointers, traverse the list all the way up to the head (root), and propagate the root information back to the elements. Note that for each element i, its successor information $s(i)$ eventually changes to root information, 5.

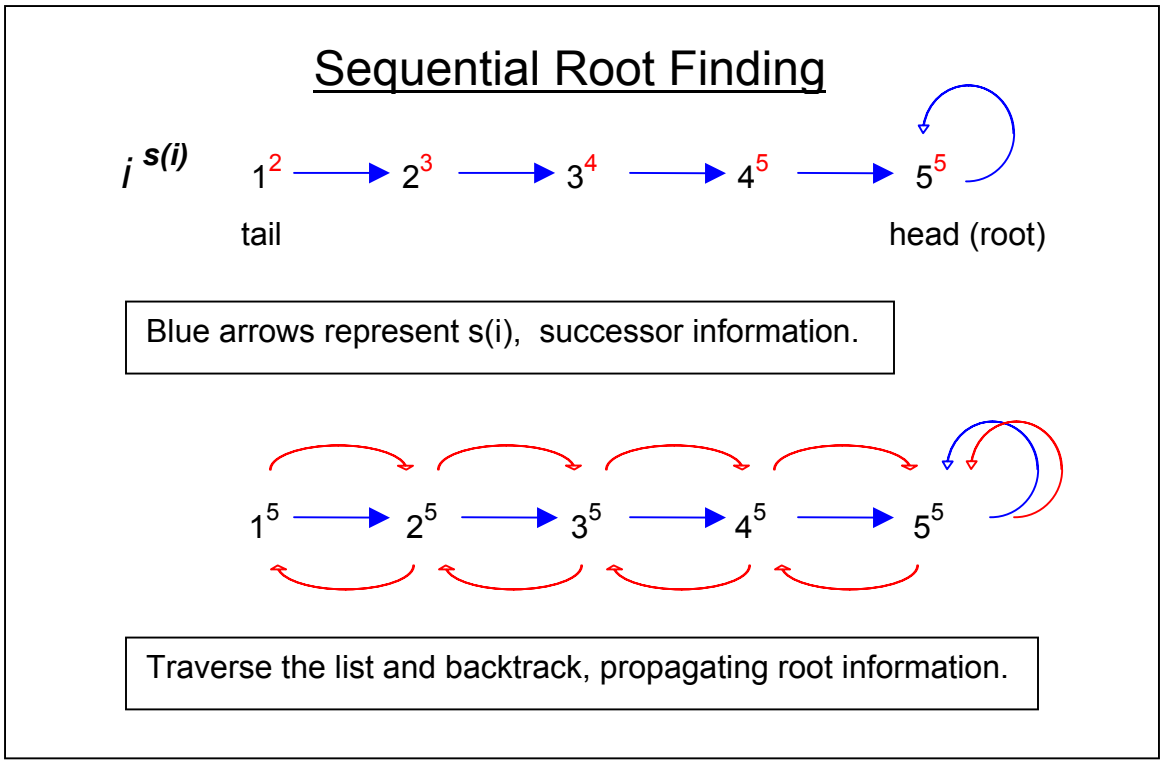

Figure 1: Root finding problem and a solution using a sequential algorithm.

On a parallel machine where the n elements of a list are randomly distributed among P processors, such a naïve sequential approach would not yield an efficient solution, especially when dealing with a huge list. An optimal parallel solution will run in $O(n / P)$ time. Pointer jumping provides a simple and elegant solution, even though it is not optimal. It solves the problem in $O(n \log n / P)$ time, paying the cost of $O(\log n)$ multiplicative factor in the running time.

Synchronized Pointer Jumping at an Abstract Level

At the abstract level, pointer jumping works as follows. Each element makes a jump onto its successor, in parallel, by following its successor pointer, reads the successor information of its successor, and uses it to update its own successor information. In at most (the ceiling of) $\log_2 n$ jumping steps, all elements obtain the root information, and pointer jumping terminates.

Traditionally, each step is synchronized. That is, an element does not proceed to the next step until all other elements have made a jump for the current step. In each step, the jumps are made concurrently and updating successor information is done concurrently. That is, even when an element jumps onto its successor that has already updated its own successor information, it will be forced to read the successor's unupdated successor information. Elements which have found the root information do not participate in future jumping steps. Figure 2 illustrates root finding on a list of five elements in three synchronized jumping steps.

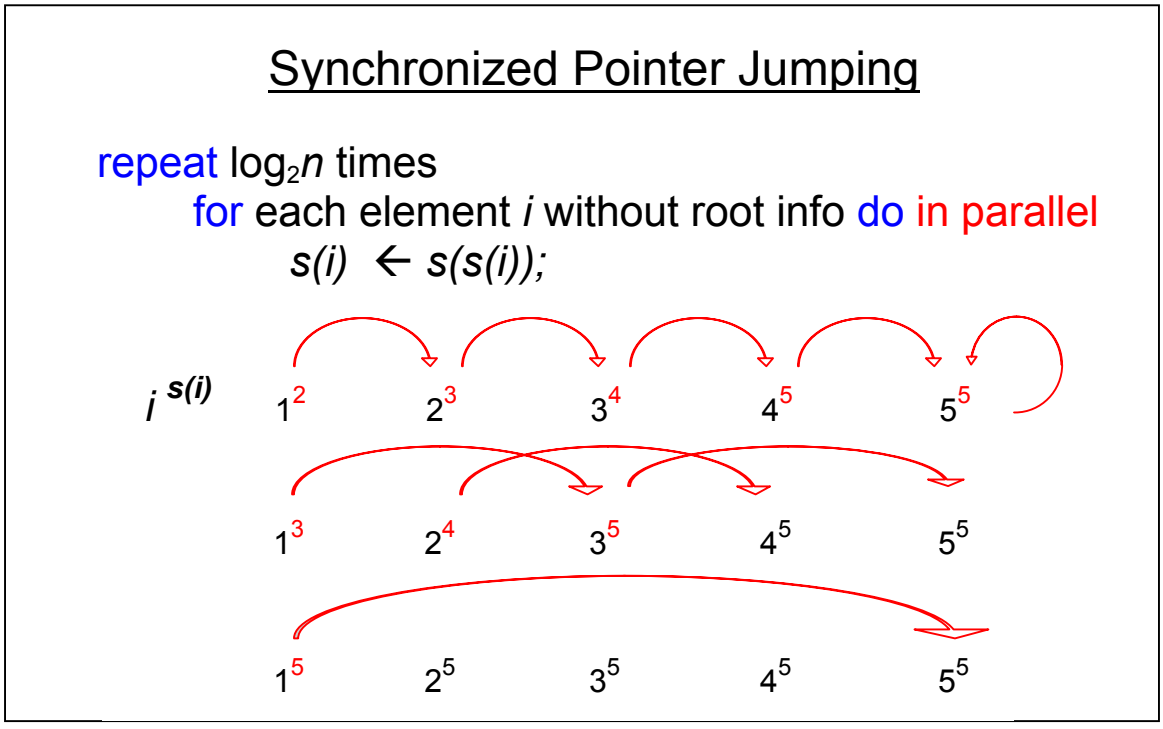

Figure 2: Root finding on a list by synchronized pointer jumping.

Synchronized Pointer Jumping on a Distributed Memory Machine

On our parallel machine model, it is the processor to which an element belongs that makes a jump for it. A jump consists of two messages. One is a *request* for successor information. It is sent to the processor that has the successor element. The other is a *reply* sent back to the processor that made the request. There are two things that are worth noting. As is the case for the elements at the abstract level, a processor does not proceed to the next step until all other processors have completed the current step. Also, updating successor information is done in such a way that each element accesses the unupdated successor information of its successor. These two things make the synchronized implementation different from the unsynchronized one.

For the root finding of Figure 2, a total of 9 jumps are made. When the successor of an element is found inside its own processor, a jump will not involve inter-processor communication. Therefore, 9 jumps do not translate into 18 messages. The expected total number of messages is *2 * total number of jumps * (P–1) / P*.

Unsynchronized Pointer Jumping

Theoretically, pointer jumping doesn't require synchronization. That is, the correctness of execution is not jeopardized by the absence of synchronization.

In the previous section we mentioned two things that make synchronized pointer jumping different from unsynchronized pointer jumping. They can be restated as follows. In unsynchronized pointer jumping, a processor that has completed its work for the current jumping step enters the next step immediately, without waiting for other processors to complete the current step. The successor information that an element gets from its successor may have already been updated.

We implemented both synchronized and unsynchronized pointer jumping on a parallel computer and observed their performance in terms of running time. We expected the unsynchronized implementation to run faster than the synchronized one for the following reasons.

1. Even though *n* elements are randomly and evenly distributed among *P* processors in the beginning, in the course of pointer jumping (and during the execution of a parallel algorithm in general) imbalance of workload among processors is likely to occur and get worse, as shown by Chung and Condon [4]. During each synchronized pointer jumping step, some processors will complete their work and idle while other processors continue to work. As the imbalance gets worse, the execution will get less efficient. By allowing processors to proceed to the next jumping step instead of idling, one might expect to achieve more efficient implementation of pointer jumping.

2. In addition, it is likely that many of the elements will reach the root in a fewer number of jumping steps (as illustrated in Figure 6), thus reducing the total number of messages.

Indeed, our unsynchronized implementation did less work, in terms of the total number of messages, as shown in Figure 3. The total number of messages is about 72 percent of the synchronized implementation, and all elements find their root information within 13 steps, compared with 16 steps of the synchronized implementation.

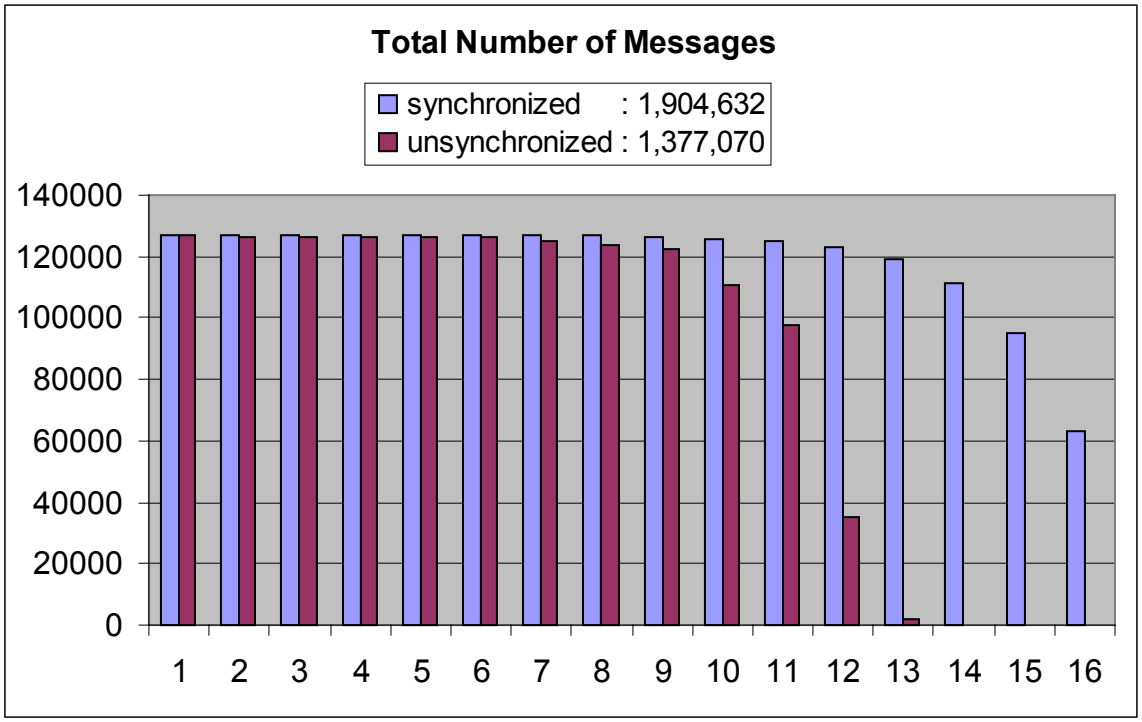

Figure 3: Total number of messages for root finding on a list of n = 65,536 elements on P = 32 processors. For synchronized pointer jumping, the x-axis represents the actual synchronized steps. For unsynchronized pointer jumping, synchronized steps are meaningless. The unsynchronized jumping activity is shown in 13 groups solely for the purpose of comparison with the synchronized implementation.

However, the unsynchronized implementation ran much more slowly than the synchronized one. Figure 4 shows that, as the number of messages becomes smaller (shown in Figure 3), the running time of the unsynchronized implementation becomes longer and longer, up to the point comparable to synchronized implementation's step 11. We decided to call it an anomaly for the reason stated in the next section.

Explanation of How the Anomaly Occurred

The primary cause of the anomaly in the unsynchronized implementation is the huge imbalance of workload among processors compared with the mild imbalance of synchronized implementation. For synchronized pointer jumping, the imbalance is not any more serious than for other parallel applications whose data are randomly and evenly distributed among processors in the beginning. For unsynchronized pointer jumping, however, the imbalance gets far worse, and at a much faster rate, as shown in Figure 5.

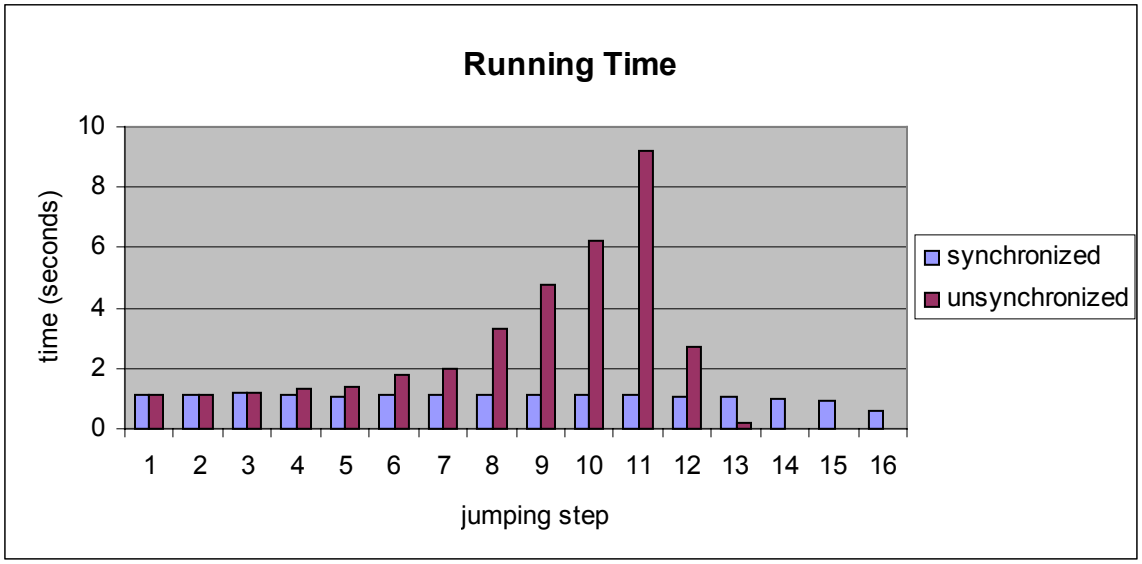

Figure 4: Running time of synchronized and unsynchronized implementations.

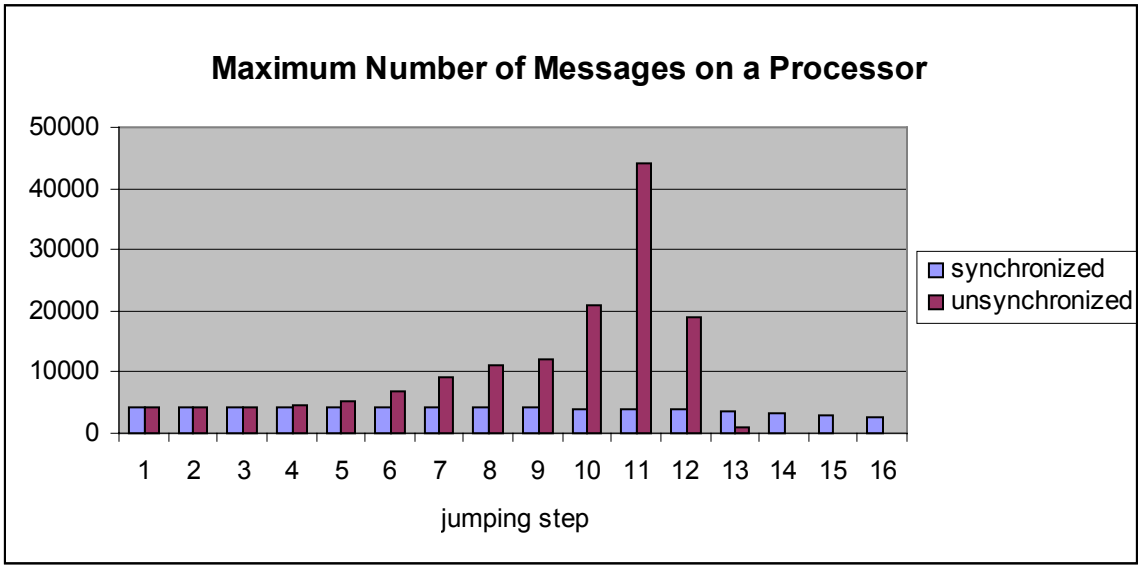

Figure 5: The maximum number of messages on a processor for synchronized and unsynchronized implementations.

For example, when unsynchronized implementation reaches a point that is comparable to step 11 of the synchronized implementation, one of the 32 processors sends a total of about 44, 000 messages. It is more than ten times the maximum number of messages on a processor in the synchronized implementation (slightly over 4,000). It is more than 40 percent of the total number of messages sent by all of the 32 processors (slightly less than 100,000, Figure 3). It means that, while this processor is doing more than 40 percent of the work, many processors idle while they wait for this processor to complete its work.

Figure 6 illustrates how imbalance is caused by the absence of synchronization. Suppose element 2 has its successor information updated (from 3 to 4) before element 1 sends a request to it. Then, when element 1 sends a request, element 2 will reply with 4, its "updated" successor information. The next jump of element 1 will be made onto element 4. Suppose, again, that element 4 already updated successor information (from 6 to 8) and replies to element 1 with that updated successor information. Then, the third jump of element 1 will be made onto element 8. In synchronized implementation, the third jump of element 1 would be made onto element 5.

In this fashion, many elements obtain the root information in fewer jumping steps than they would in the synchronized implementation. That is why the unsynchronized implementation does less work, in terms of the total number of messages, than the synchronized one, as shown earlier in Figure 3.

Then, why does the unsynchronized implementation run more slowly than the synchronized one? The answer is that the above phenomenon causes huge imbalance of workload among processors. Consider the case illustrated at the bottom of Figure 6. Elements 1, 2, 3, and 4 make jumps onto element 8. The processor that is in charge of element 8 needs to send four replies. It is possible that some of the processors in charge of elements 1, 2, 3, and 4 idle, while they wait for the replies.

This phenomenon causes the imbalance of workload to get worse and worse as unsynchronized pointer jumping proceeds. It eventually causes a few processors to do most of the work, while many others idle.

Our attempt to remove idling of processors caused more processors to idle, which in turn caused the unsynchronized implementation to run more slowly. That is why we decided to call it an anomaly. The anomaly was analogous to *Belady's anomaly*, a well-known phenomenon in operating systems: The First-In First-Out page replacement algorithm using four page frames causes, on certain data sets, more page faults than when using three page frames. The anomaly in unsynchronized pointer jumping was more rampant than Belady's anomaly. It was observed on all randomly generated data sets, not just certain data sets.

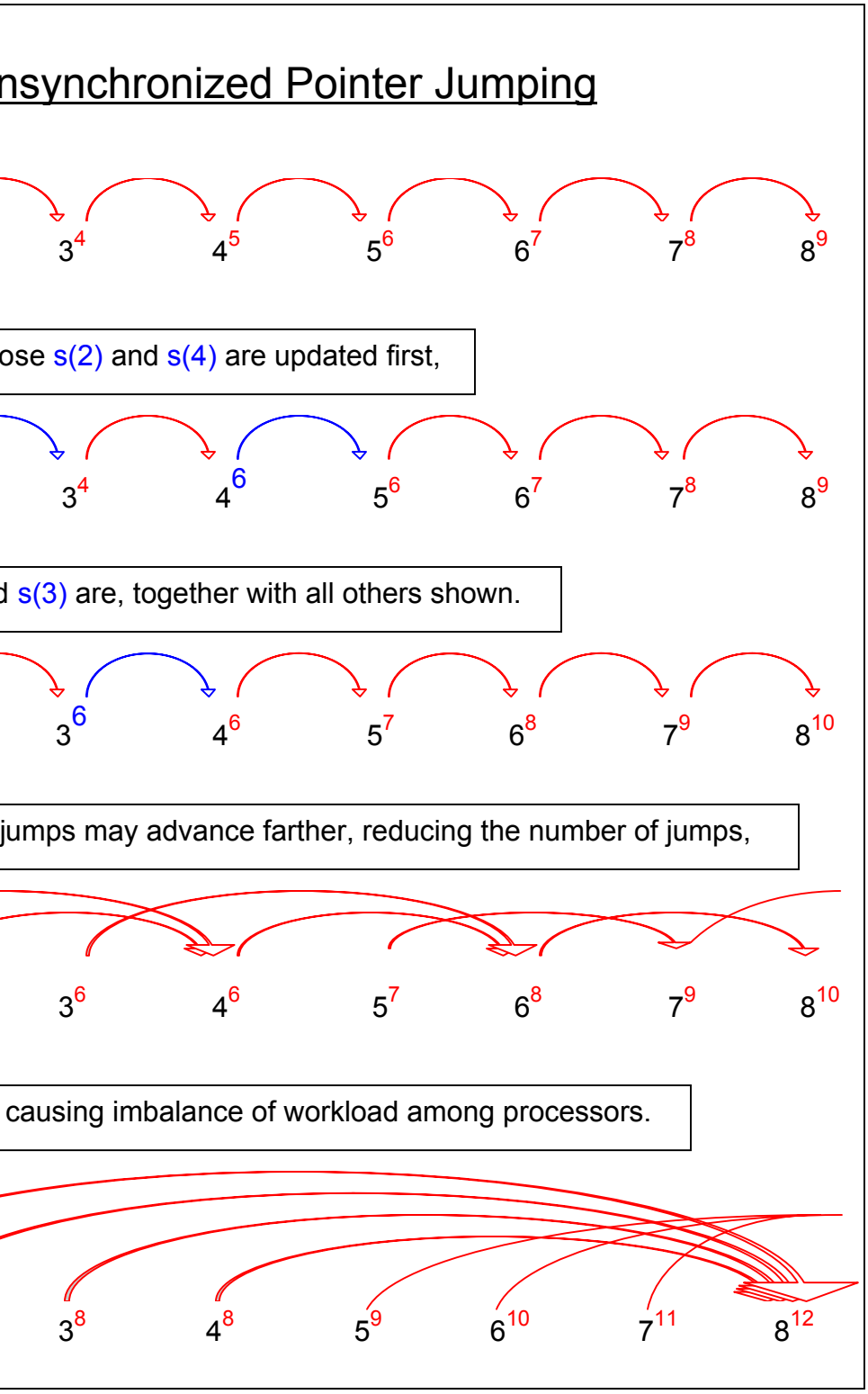

Figure 6: Unsynchronized pointer jumping.

A Special Characteristic of Synchronization

What is it that makes the workload of synchronized pointer jumping relatively well balanced? The answer is that, at each synchronized step, only one jump is made for each element which has yet to obtain the root information. There are two parts in the answer. One is "only one jump per element." It prevents a multiple number of jumps from being made onto an element. The other is that elements drop from the jumping process as soon as they obtain the root information. (If they continue to make useless jumps, then the result will be catastrophic.) In addition, a third condition mentioned earlier deserves mentioning again: Each element updates its successor information with the "unupdated" successor information of its successor. All of these factors work together to make the imbalance of workload for synchronized pointer jumping not any worse than other parallel algorithms whose data are randomly and evenly distributed among processors in the beginning.

In the beginning of the paper, we discussed a couple of general characteristics of synchronization. For pointer jumping, synchronization adds a special characteristic. Use of synchronization is justified even when it is not required for correctness. It is not an unwanted overhead but a useful mechanism that keeps the workload among processors well balanced.

Conclusion

In this paper, we presented results obtained from experimenting synchronized and unsynchronized pointer jumping on a parallel computer with distributed memory. In particular, we reported an anomaly that we observed in unsynchronized pointer jumping and explained why it occurred. Probabilistic analysis of the anomaly with regard to the imbalance of workload among processors is left for future work.

We characterized synchronization as *required* for many parallel applications and incurring *overhead*. In addition, we presented a special characteristic of synchronization for pointer jumping. It will be interesting to identify parallel algorithms to which the same special characteristic applies.

References

1. R. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. *Proceedings of the twenty-third annual ACM Symposium on Theory of Computing*, pp. 370 – 380, 1991.

2. Y. Ben-Asher. The parallel client-server paradigm. *Parallel Computing*, vol. 28, pp. 503 – 523, 2002.

3. S. Chung. Parallel design and implementation of graph algorithms for minimum spanning tree, list ranking, and root finding on trees. Ph. D. Dissertation. University of Wisconsin – Madison, 1998.

4. S. Chung and A. Condon. Parallel implementation of Borůvka's minimum spanning tree algorithm. *Proceedings of the tenth International Parallel Processing Symposium*, pp. 302 – 308, 1996.

5. R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. *Proceedings of the first annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 169 – 178, 1989.

6. P. Gibbons. A more practical PRAM model. *Proceedings of the first annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 158 – 168, 1989.

7. E. F. Grove. Connected components and the interval graph. *Proceedings of the fourth annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 382 – 391, 1992.

8. A. Krishnamurthy, S. Lumetta, D. E. Culler and K. Yelick. Connected components on distributed memory machines. *DIMACS Implementational Challenge*, 1994.

9. C. Martel and R. Subramonian. Asynchronous PRAM algorithms for list ranking and transitive closure. *Proceedings of International Conference on Parallel Processing*, vol. 3, pp. 60 – 63, 1990.

10. N. Nishimura. Asynchronous shared memory parallel computation. *Proceedings of the second annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 76 – 84, 1990.

11. A. S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.