 Guaranteeing Good Memory Bounds for Parallel Programs

F. Warren Burton

Abstract—The amount of memory required by a parallel program may be spectacularly larger than the memory required by an equivalent sequential program, particularly for programs that use recursion extensively. Since most parallel programs are nondeterministic in behavior, even when computing a deterministic result, parallel memory requirements may vary from run to run, even with the same data. Hence, parallel memory requirements may be both large (relative to memory requirements of an equivalent sequential program) and unpredictable.

Assume that each parallel program has an underlying sequential execution order that may be used as a basis for predicting parallel memory requirements. We propose a simple restriction that is sufficient to ensure that any program that will run in $n$ units of memory sequentially can run in $mn$ units of memory on $m$ processors, using a scheduling algorithm that is always within a factor of two of being optimal with respect to time.

Any program can be transformed into one that satisfies the restriction, but some potential parallelism may be lost in the transformation. Alternatively, it is possible to define a parallel programming language in which only programs satisfying the restriction can be written.

1 INTRODUCTION

If recursion and dynamic memory allocation are permitted in a parallel programming language, then it is often difficult to control or predict how much memory a program will require. In part, this is due to the fact that parallel programs are usually nondeterministic in their behavior, even when they are deterministic in their semantics.

If a program contains more parallelism than a parallel computer system can support, then from time to time a scheduler must decide which of several processes to run next. Often one choice will result in memory being released by one process while another choice will result in more memory being allocated by a different process. The memory required by a parallel program is likely to depend on the order in which various processes are scheduled, which in turn may depend on the number of processors available, what else is being done by a computer system when a particular program runs, etc. This means that even experimentation is not sufficient to determine whether a parallel program will run within a given memory bound, because it might today but not tomorrow.

There are really two related problems we wish to address:

1) Finding a simple basis for predicting the memory requirements of a parallel program, and
2) Ensuring that this prediction holds for all valid executions of the program.

We will assume that each parallel program has a unique and known behavior when it is run on a single processor. We will use the sequential memory requirements of a program as a basis for predicting the parallel memory requirements. For the most part, we will assume that programs have deterministic semantics, but will briefly consider nondeterminism in Section 6. We will also assume that the memory required by a program when run sequentially can be determined, either by analysis or experimentation. Our goal will be to ensure that any program that will run sequentially in $n$ units of memory will run in $mn$ units of memory on $m$ processors. We have now solved the first problem by making the second problem more specific.

It is known that there exist parallel programs that cannot be scheduled in such a way as to have good time and good memory performance at the same time [1]. Therefore, to solve the second problem we will need to place a restriction on the class of programs that are permitted. This restriction is given near the end of Section 2. In addition, some restrictions must be placed on the order in which work can be scheduled. Scheduling is considered in Section 5.

Our approach is to associate memory and processors, and to show that the memory associated with any one processor at any point in time is a subset of the memory that would have been allocated at some point during sequential execution of the program. The total amount of memory associated with all processors, at any point in time, may be more, but never less, than the amount of memory that is actually in use.

It always is possible to transform a parallel program into one that satisfies our restriction, but sometimes this will result in loss of parallelism. (This is trivially true. In the worst case we add synchronization constraints to the parallel program until all parallelism is lost and we are left...
with the sequential program that is the basis for our predicted memory requirements. Alternatively, by restricting ourselves to certain language features, we can ensure that all programs meet our restrictions. For example, a deterministic, block structured language with recursion and \texttt{begin-coend} constructs will meet our requirements, provided all memory allocated within a block, either on the stack or in the heap, is released when the block is exited. Nondeterminism, which would arise if shared variables may be read and modified in parallel, will be considered briefly in Section 6.

Given a program and a scheduler that satisfy our restrictions, the time required by the program will be within a factor of two of the time required by the program with an optimal scheduling algorithm. (We note that optimal scheduling requires advanced knowledge of the amount of time each process will require, and other information not available at the time scheduling decisions must be made. Furthermore, optimal scheduling is NP-hard even with this information.) Clearly, optimal scheduling cannot be implemented in practical systems.

We should note two limitations to our method. First, we assume that if there is work to be done, then an idle processor can find work to do within bounded time. This is the case if there is a central pool of available work, but this may be a bottleneck in a large system with fine grain parallelism. If available work is distributed over numerous processors, then the worst-case time required for an idle processor to locate available work may increase with larger numbers of processors. A partial solution to this problem is discussed in the section on related work, Section 7.

A second limitation to our method is that we do not consider the problem of memory fragmentation that can result in the case of blocks of memory of differing size being allocated and released in an arbitrary order. We consider only the total amount of memory required at any given time. One partial solution to this problem, in a limited context, is mentioned in Section 4.

In principle, our approach could work for a computer system with distributed memory. However, communication costs may be a problem. This and other problems related to memory management in distributed systems, while important, are largely orthogonal to the issues considered in this paper. For this reason, we will assume a shared memory model. (We will also ignore memory contention and other problems that this simplifying assumption may introduce.)

2 THE MODEL

In this section, we will present a theoretical model for the time and memory requirements of a broad class of parallel programs, and present some initial results. Later we will show how this model relates to practical programming languages, and then, using the model, show how good time and memory performance can be guaranteed for an important class of parallel programs.

We will assume that the execution of a parallel program consists of performing a collection of actions. Each action can allocate or deallocate memory and/or run for a period of time. If an action does both, we will assume that any memory allocation or deallocation is done when the action starts. This is not a significant restriction, because memory allocation and deallocation at other times can be modeled by replacing the single action with a sequence of actions.

Synchronization constraints may determine when an action can be executed, but once an action starts it cannot be subject to any external synchronization constraints. While we will not use preemptive scheduling of actions, there is nothing in our model that would prohibit this.

An action is a run-time concept. For example, an action could model the execution of an instruction, but not the instruction itself. In this case, there would be a separate action for each time the instruction is executed. We will not be concerned with exactly what an action models (e.g., the execution of a machine instruction, a block of instructions, or perhaps a procedure body containing loops and conditional but no synchronization constructs or statements allocating memory).

We will group actions into stints, which may be nested. Each stint will consist of a collection of actions that may be executed sequentially as a thread of computation. Alternatively, in a stint containing nested stints, some actions within the stint may be executed in parallel. The first action in a stint is called a start action. The key property of a start action is that once the start action can be executed, every other action in the stint can be executed (after previous actions in the stint) without delays caused by waiting for actions outside the stint to be run.

A stint is said to be memory-limited if it releases at least as much memory as it allocates. We will show that if every stint in a parallel program is memory-limited, then it is possible to execute the program in parallel within reasonable time and memory bounds. These terms will be defined more formally below.

In the model, we will not distinguish between different types of memory. We may assume that the memory for the program code and all global information is allocated before the execution of a program starts, so this memory need not be considered in our model. Stack and heap memory may be allocated from a common pool without distinction. Alternatively, our results may be applied to heap and stack spaces separately.

\textbf{Definition. (Time and Memory Task System or TAM System)}

A time and memory task system or TAM system is a five-tuple of values, \( T = (A, <, t, \delta) \), where \( A \) is a set of actions, \(< \) is a partial order defined on those actions, \(< \) is a total order containing \(< \), \( t \) is a function from \( A \) to nonnegative reals, and \( \delta \) is a function from \( A \) to integers, with two restrictions given below.

A time and memory task system is used to model the execution of a parallel program. The partial order, \(< \), defines all synchronization constraints on a parallel program execution. If \( a < b \) then \( b \) cannot be started until \( a \) has been completed. The pair \((A, <)\) is often called a task system [2]. Actions are often called tasks [3], or subtasks [4]. We prefer the term action, since the term task in computer science usually denotes a collection of actions.
The total order, $<$, defines the order in which actions would be executed on a single processor (i.e., with sequential execution). A total order may be used in list scheduling [2] or as a basis for expected memory requirements [5]. As usual, $\leq$ will denote the reflexive closure of $<$. 

For any $a$ in $A$, $t(a)$ is the time required to execute $a$, and $\Delta(a)$ is the amount of memory allocated when $a$ is executed. If $\Delta(a)$ is negative, then memory is deallocated. The memory allocation or deallocation occurs when $a$ starts execution.

We should mention two limitations to this model. First, parallel programs are assumed to be deterministic. One consequence of this is that memory management based on reference counting or garbage collection is not allowed since the amount of memory released by an action, which may depend on whether other actions requiring an item of memory have been run, will be nondeterministic in some cases. Second, the model excludes some reasonable parallel control structures. For example, with lazy evaluation in a functional programming language, an expression may be evaluated as soon as any one of several possible processes is known to require its value. This corresponds to executing an action after any ancestor (rather than all ancestors) has been run. Similar structures may be found in other types of parallel languages, but often involve nondeterminism, which we have already excluded.

We will sometimes represent a TAM system using a diagram. Unless stated otherwise, all actions will require unit execution time in these examples. (That is, $\forall a \in A, t(a) = 1$.) This restriction applies only to the examples we give, and not to the model. Each action will allocate zero or one units of memory, or release or one unit of memory. Again, this simplifying restriction applies only to our examples. Actions will be represented by squares, with a shaded bar at the top if the action allocates a unit of memory ($\Delta(a) = 1$), a shaded bar at the bottom if the action releases a unit of memory ($\Delta(a) = -1$), and no shaded bar otherwise ($\Delta(a) = 0$). If we are not concerned with how much memory an action allocates, then the action will always be shown without a shaded bar. The $<$ relation will be the transitive closure of the relation represented by solid arrows, and the $\leq$ relation will be the transitive closure of the relation represented by all arrows, solid or dashed.

For example, Fig. 1 represents a TAM system with three actions, $a_1$, $a_2$, and $a_3$. The actions $a_1$ and $a_2$ may be executed in parallel, but both must be executed before $a_3$. With sequential execution, $a_1$ will be executed before $a_2$. All actions require one unit of time. Action $a_1$ allocates a unit of memory, which is released by $a_3$.

In the remainder of this paper, $A = \{a_1, a_2, \ldots, a_n\}$, with $a_i < a_{i+1}$ for $0 \leq i < n$, will denote the entire set of actions in a TAM system.

**Definition.** (Net Memory Allocation) For any subset $B$ of $A$, the net memory allocation by $B$ is $\Delta(B) = \sum_{a \in B} \Delta(a)$.

The net memory allocation may be negative if $B$ frees memory that was previously allocated.

**Definition.** (Prefix) $B$ is called a prefix of $A$ if $B \subseteq A \land (\forall b \in B, \forall a \in A. (a < b \Rightarrow (a \in B)))$, where $\Rightarrow$ denotes logical implication.

When a TAM system is executed in parallel, the set of actions completed at any point in time will be a prefix. That is, if a prefix contains an action, $b$, then it must also contain any action, $a$, in $A$, that must be executed before $b$.

![Fig. 1. A simple TAM system.](image)

To avoid special cases later, we will require that a program eventually deallocate all allocated memory. This is not an important restriction, since it may be satisfied by releasing all remaining memory at the time a program terminates. We will also require that the total amount of memory allocated by a program at any one time is nonnegative.

**Restriction 1.** $\Delta(A) = 0$.

**Restriction 2.** If $B$ is a prefix of $A$, then $\Delta(B) \geq 0$.

These restrictions do not ensure good memory performance. That restriction is given below in the definition of "memory-limited."

In the model, there is no correspondence between the allocation of a particular unit of memory and the release of that specific unit. For example, if we look ahead to the TAM system shown in Fig. 3 the memory allocated by action $a_1$ might be released by $a_2$ or $a_3$. This information is outside the scope of the model. All we know is that $a_1$ and $a_2$ each allocate a unit of memory and $a_3$ and $a_4$ each release a unit of memory. (Clearly, the memory allocated by $a_1$ must be released by $a_2$.) The significance of the thick black squares and gray rectangles with rounded corners in the figure will be explained later.

The model assumes that the parallel program execution is deterministic. The same set of actions must be produced, with the same partial order and total order, and with each action requiring the same time and executing the same memory allocation, regardless of when the actions are actually scheduled. This limitation, and ways around it, are discussed in Section 6.

We will now present a series of definitions leading to the definition of a memory-limited TAM system. A *stint* is a collection of actions having particular properties and a memory-limited TAM system is a TAM system in which the net memory allocation by any stint is at most zero. In Section 5, we will present a scheduling algorithm yielding the performance we desire for memory-limited TAM systems.

In the following definitions, recall that we are letting $A$ denote the entire set of actions in a TAM system.
DEFINITION. (Min, Max, Pred, Succ) If B is a nonempty subset of A, then \( \text{min}(B) \) is that action \( b \in B \) such that for any other action \( c \in B, b < c \). Since \( < \) is a total order, \( \text{min}(B) \) is unique. The function \( \text{max} \) is similarly defined. For any action \( a \neq \text{min}(A) \), the action \( \text{pred}(a) \) is the action immediately before \( a \) (i.e., \( \text{pred}(a) < a \) but there is no action \( b \in A \) such that \( \text{pred}(a) < b < a \)). Similarly, \( \text{succ}(a) \) is that action immediately after \( a \), provided \( a \neq \text{max}(A) \).

Informally, \( \text{min}(B) \) would be the first action in \( B \) to be executed if the entire TAM system were executed sequentially, and \( \text{max}(B) \) would be the last action in \( B \) to be executed sequentially. The actions \( \text{pred}(a), a \) and \( \text{succ}(a) \), when defined, will always be executed consecutively with sequential execution.

DEFINITION. (Chain) A chain is a set of consecutive actions with respect to the total order \( < \). That is, \( C \subseteq A \) is a chain if and only if \( \forall a \in A, \text{min}(C) < a < \text{max}(C) \Rightarrow a \in C \).

The definition of a chain imposes no restrictions on the partial order, \( < \), of the actions in the chain. In fact, since any partial order can be extended to a total order, actions in a chain can have any partial order. The same is true of free chains defined below.

We are particularly interested in chains that can be executed sequentially once the first action in the chain can be executed.

DEFINITION. (Free Chain) A free chain, \( C \), is a chain where each action can be executed as soon as all of the previous actions in the chain have been completed. That is, a chain \( C \) is a free chain if and only if \( \forall c \in C, \forall a \in A, (a < c \land a \notin C) \Rightarrow a < \text{min}(C) \).

If \( c \in C \) for some free chain \( C \) and \( a \) must be executed before \( c \) (i.e., \( a < c \)) for some action \( a \in A \), then either \( a \) must be in \( C \) (with \( a < c \) since \( a < c \)) or must be executed before \( C \) can start (i.e., \( a < \text{min}(C) \)). Clearly, once the first action in a free chain can be executed, all other actions in the sequence can be executed sequentially, without having to wait for any action outside the sequence to complete.

In terms of our diagrams, a solid arrow originating outside a free chain, \( C \), cannot point to any action in \( C \) except \( \text{min}(C) \). However, a solid arrow can connect two actions within \( C \).

A free chain may contain internal parallelism and synchronization. For example, the entire sequence of actions in Fig. 2 is a free chain. (The thick black squares and gray rectangles will be explained shortly.) Notice that a subsequence of a free chain may or may not be a free chain. For example, \( \{a_3, a_2\} \) and \( \{a_7, a_8\} \) are free chains but \( \{a_4, a_6, a_8, a_7\} \) is not. An entire TAM system is always a free chain, since it can always be executed sequentially.

Clearly the definition of a free chain imposes no restrictions on the internal structure of the actions with respect to the partial order, \( < \), since any partial order, \( < \), can be used in a TAM system.

We will require that each possible thread of computation start with an action having properties specified in the following definition. (We are using "thread of computation" as an informal alternative to the term "stint," which is defined below.)

DEFINITION. (Start Action) An action, \( a \), is called a start action if and only if \( a = \text{min}(A) \lor \text{pred}(a) \neq a \lor (\exists b \in A, b < a \land b \neq \text{pred}(a)) \).

Any action \( a \) is a start action unless the completion of \( \text{pred}(a) \) is a necessary and sufficient condition for \( a \) to be started. Any parallel thread of computation will start with a start action. Let us consider the three cases given in the definition in more detail. Clearly \( \text{min}(A) \), the first action of the program, must be a start action, since \( \text{pred}(\text{min}(A)) \) does not exist. If \( \text{pred}(a) \neq a \) then \( a \) may be started before \( \text{pred}(a) \) has been completed, thereby starting a new thread. Finally, if \( \exists b \in A, b < a \land b \neq \text{pred}(a) \) then in the case where \( b \) has not been executed it is possible that a thread of computation will end when \( \text{pred}(a) \) is completed, but a new thread starting with \( a \) may be initiated when \( b \) is completed.

In Fig. 1, the actions \( a_1, a_2 \) and \( a_3 \) are all start actions. The action \( a_1 \) qualifies as a start action because \( a_1 = \text{min}(A) \). Since \( a_1 = \text{pred}(a_2) \neq a_2, a_2 \) is also a start action, so \( a_2 \) and \( a_3 \) may be executed in parallel. Finally, \( a_3 \) is a start action since the completion of \( a_2 \), which may happen before the completion of \( a_3 \), is not sufficient for \( a_3 \) to be started.

In terms of our diagrams, any action \( a \), other than \( \text{min}(A) \), is a start action if and only if either the arrow from \( \text{pred}(a) \) to \( a \) is dashed or there is a solid arrow to \( a \) from another than \( \text{pred}(a) \), or both.

Returning to Fig. 1, \( B = \{a_2, a_3\} \) is not a free chain because \( a_1 < a_3 \) and \( a_1 \notin B \) but \( a_1 < \text{min}(B) = a_2 \). Hence, \( a_1 \) can block the execution of \( B \) by delaying the execution of \( a_3 \) after the execution of \( B \) has started with \( a_2 \). On the other hand, \( \{a_1, a_3, a_2\}, \{a_1, a_2\}, \{a_1\}, \{a_2\} \) and \( \{a_3\} \) are free chains.

DEFINITION. (Stint) If \( a \) is a start action, then \( \text{stint}(a) \) is the largest free chain, \( B \), such that \( a = \text{min}(B) \). Any free chain, \( B \), is called a stint if \( B = \text{stint}(a) \) for some start action \( a \).
Fig. 2 shows a TAM system with all start actions indicated by thick black squares, and with all \( \textit{stints} \) enclosed in gray rectangles with rounded corners. \( \textit{Stints} \) are nested, and \( a_{e} \) and \( a_{s} \) are trivial \( \textit{stints} \) with only one action in each. Notice that there is a one-to-one correspondence between \( \textit{stints} \) and start actions, with each start action being the first action of the corresponding \( \textit{stint} \). Once the start action of a \( \textit{stint} \) can be executed, execution can continue sequentially with all of the other actions in the \( \textit{stint} \), without any possible delay caused by waiting for actions outside the \( \textit{stint} \) to be completed. Each \( \textit{stint} \) is maximal with respect to this property. On the other hand, in the case of nested \( \textit{stints} \), it may be possible for some actions within a \( \textit{stint} \) to execute while the start action is delayed waiting for actions outside the \( \textit{stint} \) to be completed. For example, \( a_{s} \) may be executed before \( a_{e} \) can be.

Notice that the action following a \( \textit{stint} \) is always the start action of another \( \textit{stint} \).

We will finish this section with several definitions and results that will be used in Sections 3 and 5, where composition of \( \textit{stints} \) and scheduling of programs will be considered.

**Lemma 1.** If \( C \) is a \( \textit{stint} \) and if \( d = \text{succ}\textit{max}(C) \) exists, then \( \exists a \in A \), \( a < d \land a \notin C \land a < \text{min}(C) \).

**Proof.** If this were not the case, then by the definition of a free chain, \( C \cup \{d\} \) would be a free chain, so \( C \) would not be maximal and hence would not be a \( \textit{stint} \).

**Theorem 2.** \( \textit{Stints} \) can be properly nested, but cannot overlap otherwise.

**Proof.** Fig. 2 demonstrates that \( \textit{stints} \) can be nested.

Assume that \( B \) and \( C \) are two overlapping, but not nested, \( \textit{stints} \), with \( \text{min}(B) < \text{min}(C) \leq \text{max}(B) < \text{max}(C) \). Let \( d = \text{succ}\textit{max}(B) \) so \( d \notin C \) but \( d \notin B \). By Lemma 1 there must exist an action \( a \) such that \( a < d \land a \notin B \land a < \text{min}(B) \). Clearly \( a < \text{min}(B) \) and \( a \notin C \).

Now we have two cases to consider. If \( a < \text{min}(C) \) then, by the definition of free chain, \( \text{min}(C) \) cannot be in the \( \textit{stint} \) \( B \) (i.e., \( a < \text{min}(C) \land a \notin B \) but \( a < \text{min}(B) \)), which contradicts our assumption. On the other hand, if \( a < \text{min}(C) \) then, by the definition of free chain, \( d \) cannot be in the \( \textit{stint} \) \( C \) (i.e., \( a < d \land a \notin C \) but \( a < \text{min}(C) \)), which again is a contradiction.

Hence, \( \textit{stints} \) cannot overlap unless they are nested.

**Definition.** (Initial Sequence) If \( B = \{a_{1}, a_{s}, \ldots, a_{k}\} \) is a \( \textit{chain} \), then \( \{a_{1}, a_{s}, \ldots, a_{k}\} \) for \( i - 1 \leq k \leq j \) is called an initial sequence of \( B \).

Note that an initial sequence may be empty when \( k = i - 1 \).

**Definition.** (Maximum Sequential Memory Requirement) The maximum sequential memory requirement of a \( \textit{chain} \) \( B = \{a_{1}, a_{s}, \ldots, a_{k}\} \) is:

\[
M(B) = \max \{\Delta(C) \mid C \text{ is an initial sequence of } B\}
\]

The maximum sequential memory requirement is the maximum amount of memory allocated by the chain at any one time with sequential execution. This may be zero, even if memory is allocated, provided the chain releases memory before allocating more memory. The maximum sequential memory requirement of a chain is always at least zero, since the net allocation by the empty initial sequence is zero.

The following lemma states that the maximum sequential memory requirement of an entire program must be at least as large as the maximum sequential memory requirement of any chain contained in the program, which seems reasonable, since the only alternative would be to have a negative amount of memory allocated when the chain is started in the context of sequential execution of the overall program. This lemma will allow us to show that by keeping the number of active chains bounded during a parallel execution of a program we can keep the parallel memory requirements bounded as well.

**Lemma 3.** For any \( \textit{chain} \), \( C \), contained in the TAM system with actions \( A_{1}, \Delta(C) \leq M(A) \).

**Proof.** Let \( C = \{a_{1}, a_{s}, \ldots, a_{k}\} \) and \( B = \{a_{1}, a_{s}, \ldots, a_{k}\} \).

Since \( B \cup C \) is an initial sequence of \( A_{1} \), \( \Delta(B) \cup C \leq M(A) \) by the definition of \( M(A) \). Since \( B \) and \( C \) are disjoint, \( \Delta(B) \cup C = \Delta(B) + \Delta(C) \), so \( \Delta(B) + \Delta(C) \leq M(A) \).

Finally, by Restriction 2, \( \Delta(B) \geq 0 \), so \( \Delta(C) \leq M(A) - \Delta(B) \leq M(A) \).

**Definition.** (Memory-limited) A free \( \textit{chain} \), \( B \), is a memory-limited \( \textit{chain} \) if \( \Delta(B) \leq 0 \). A TAM system is a memory-limited TAM system if every \( \textit{stint} \) in the TAM system is a memory-limited \( \textit{chain} \).

The restriction to memory-limited TAM systems is the restriction we need to get the performance we desire. In Section 5 we will see that any memory-limited TAM system can be scheduled to ensure that any program that will run in \( n \) units of memory sequentially can run in \( mn \) units of memory on \( m \) processors, using a scheduling algorithm that is always within a factor of two of being optimal with respect to time.

Any TAM system can be transformed into a memory-limited TAM system by strengthening the relation, but sometimes at the cost of some loss of parallelism. This is trivially true, since in the worst case the partial order can be strengthened to be identical to the total order.

A \( \textit{stint} \) may not need to release all of the memory that it allocates, in order to be memory-limited, provided it releases some memory that it has not allocated. Fig. 3 shows a memory-limited TAM system with all start actions indicated by thick black squares, and with all \( \textit{stints} \) enclosed in gray rectangles with rounded corners. Notice that \( a_{s}, a_{e} \) forms a \( \textit{stint} \). However, the memory released by \( a_{e} \) must have been allocated by \( a_{s} \) or \( a_{e} \) and not \( a_{e} \), since \( a_{e} \) may be executed before \( a_{e} \). The \( \textit{stint} \) \( a_{e} \) releases memory without allocating any.

The model we have proposed is very general. In practice, an attempt to ensure that a TAM system is memory-limited may result in the imposition of stronger restrictions. For example, a \( \textit{stint} \) may be required to release all of the memory that it allocates, not just an equivalent amount.

### 3 Composing Memory-Limited TAM Systems

In this section, we will show one method for composing memory-limited TAM systems. This result will be used in the following section where several related memory-limited programming languages will be presented.
**Theorem 4.** If $T_1$ and $T_2$ are memory-limited TAM systems and $a$ is an action of $T_2$ for which $a < \text{suc}(a)$ and $\delta(a) = 0$, then $T_3 = (T_1/a|T_2)$ is a memory-limited TAM system.

**Proof.** To show that $T_3$ is memory-limited we must show that for each $\text{stint}$, $C$, in $T_3$, $\Delta(C) \leq 0$. Let $C$ be any $\text{stint}$ in $T_3$, and let $c = \text{min}(C)$. Hence, $C = \text{stint}_3(c)$, where the subscript indicates the TAM system in which the function $\text{stint}$ is being used. We have several cases to consider:

If $C \subseteq A_1$, then $C$ is also a $\text{stint}$ in $T_1$. Since $T_1$ is memory-limited, $\Delta(C) \leq 0$. Similarly, if $C \subseteq A_2$ then $C$ is a $\text{stint}$ in $T_2$ and $\Delta(C) \leq 0$.

In the remaining cases, $C$ includes actions from both $T_1$ and $T_2$.

If $c$ is in $A_2$ then $a \in \text{stint}_2(c)$. (If this were not so, then it would be the case that $\text{stint}_3(c) = \text{stint}_2(c)$ and $C$ could contain no actions from $T_1$.) Since $A_1$ is a free chain, all of $A_1$ must be in $C$. That is, 

$$\text{stint}_3(c) = A_1 \cup \text{stint}_2(c) \setminus \{a\}$$

Hence

$$\Delta(\text{stint}_2(c)) = \Delta(A_1) + \Delta(\text{stint}_2(c)) - \delta(a) = \Delta(\text{stint}_2(c)) \leq 0$$

since $\Delta(A_1) = \delta(a) = 0$.

In the case $c$ is in $A_1$ and $C$ includes at least one action in $A_2$. Since $\text{suc}_2(a)$ is the first action in $T_2$ following those from $T_1$, it must be in $C$. Recall that $a <_3 \text{suc}_2(a)$. By the definition of $<_3$, $b <_3 \text{suc}_2(a)$ for every $b$ in $A_1$, so $A_2 \subseteq C$. (Otherwise, an action in $A_1$ but not in $C$ could delay the execution of $\text{suc}_2(a)$.) It follows that $C = A_1 \cup \text{stint}_2(c) \setminus \{a\}$. Hence

$$\Delta(C) = \Delta(A_1) + \Delta(\text{stint}_2(c)) - \delta(a) = \Delta(\text{stint}_2(c)) \leq 0$$

since $\Delta(A_1) = \delta(a) = 0$.

**Definition.** (Slot) A slot is any action, $a$, in a memory-limited TAM system, such that $a < \text{suc}(a)$ and $\delta(a) = 0$.

A memory-limited TAM system can be substituted for any slot in another memory-limited TAM system to produce a third.

### 4 Some Simple Memory-Limited Languages

Are the results presented in the preceding sections of any practical value? In this section we outline several types of parallel programming languages that are powerful enough to use for a broad class of realistic applications, while supporting memory-limited programming, thereby answering this question in the affirmative.

In this section, we will assume that all programs are fully deterministic. We will not consider language restrictions necessary to enforce this restriction (e.g., Ensuring that shared variables are not modified). In practice it may be better to allow nondeterminism, but with the understanding that programmers introduce nondeterminism at their own risk. See Section 6.

We will start by considering two kinds of TAM systems. Each is rather boring by itself. Programs become more interesting when these are composed, as described in the previous section.
Kind I. Any purely sequential TAM system must be memory-limited, since, by Restriction 1 from Section 2, its only stint must release all allocated memory.

Kind II. Any TAM system that allocates memory only at the start of execution must be memory-limited, since
- The top level stint (the entire program) is memory-limited by Restriction 1.
- Any other stint can only release memory and hence is memory-limited.

Note: The initial memory allocation may be done in a sequence of actions rather than a single action.

A TAM system of either kind may contain zero or more slots.

Definition. (Component). A component will be any TAM system of either of the above kinds.

Language I consists of all programs that can be constructed by starting with a component of either of the above kinds and substituting components of either kind into slots, with nesting to any depth (including an indeterminate depth that might result from the use of recursion). Recall that actions in a TAM system model only the time required and the memory allocated by a program. In a programming language, the source code corresponding to a run-time component of either kind may be allowed to access variables declared in an outer component. It is the allocation and release of memory that must be tied to a specific component.

Before going on, we should note that not all useful memory-limited TAM systems can be generated by composing components. For example, general strict computations as defined in [6] are useful and are memory-limited, assuming a suitable (depth first) total order, but cannot in general be generated by composing memory-limited TAM systems of the two kinds described above.

A very simple memory-limited language, which we shall call Language II, will contain only components of the first kind that allocate all memory at the start of execution (so they are actually components of the second kind as well) and components of the second kind that are identical to the TAM system in Fig. 4. This produces a simple block structured language of the type that would need only a run-time stack for sequential execution and has only a cobegin-coend construct for parallelism. This language may include recursion.

We are informally mixing compile time and run-time concepts here. We should note that a cobegin-coend construct in a source program may generate a large number of cobegin-coend components in the run-time TAM system if the source level cobegin-coend occurs within a loop or recursive procedure. We will continue this informal mixing of compile time and run-time constructs. It should be clear how they relate, usually with the only important difference being that a single source level construct may correspond to any number of run-time components.

Brich Hansen [7] has proposed a method for avoiding the memory fragmentation that results from the allocation and deallocation of stack frames of differing sizes in parallel programming languages supporting recursion. His method is to have a different pool of memory for each different stack frame size. The size of each pool, determined at run-time, is exactly large enough to hold the maximum number of stacks frames of the size that may be allocated at any one time. With Language II, this can be used with our scheduling algorithm without a problem. Of course, both sequential and parallel memory requirements will usually increase.

![Diagram](image)

Fig. 4. A cobegin-coend TAM system.

Language III is not a subset of Language I. In Language III all components of Kind I but only the cobegin-coend construct of Kind II are allowed. Components of Kind I may allocate and release memory from a heap in an arbitrary manner. We allow ordinary composition as described above, plus a second form of composition.

Let the statement $S_1$ be any source level (possibly compound) statement in a program, $P$, in Language III, and let $S_2$ be any memory-limited statement. We can substitute

```
cobegin
  S_1;
  S_2;
coend
```

for $S_1$ in $P$, and still have a memory-limited program since any stint that contained $S_1$ in the original program will also contain $S_1$ (now in slot of the cobegin-coend construct) in the resulting program. This means that we have more freedom to allocate or release heap memory in the left arm of a cobegin-coend construct than in the right arm.

We can extend Language II in a different way to allow heap memory allocation in components of Kind I, provided all heap memory is released when the component is exited.

In Language IV, components of either kind can allocate a heap pool of some fixed size in the initial memory allocation. Within the component and within any nested component, heap memory may be allocated from a heap pool. This allows us to export dynamically allocated memory from an inner component to an outer component, at the cost of previously reserving sufficient memory in an outer component. To avoid the nondeterminism that would result if the
success of an allocation from a heap pool depends on whether previously allocated memory from the heap pool has been released, we will probably want to place the restriction that no memory in a heap pool is reclaimed until the component in which the pool was allocated terminates. (An alternative solution to avoiding this form of nondeterminism, which will not generalize to Language V below, is to prohibit more than one arm of a cobegin-coend construct from allocating or releasing memory from a given pool.) In either case, all memory in the heap pool must be released when the component that created it terminates.

We can extend our Language IV further to allow arbitrary collections of _processes_ to communicate via communication channels of bounded size (i.e., bounded buffers) and to synchronize in arbitrary ways, yielding Language V. We do this by allowing components of Kind II to have arbitrary partial orders for slots and requiring that the memory for the communication channels be allocated when a component of Kind II is initialized. Each slot may be replaced either by a component of Kind II with a similar structure, or Kind I, which in turn may contain components of Kind II communicating via local communication channels. A source level process corresponds to the actions of a subset of the slots of a component of Kind II (including nested components). We omit the details. No process can retain any memory that it has allocated at any point where it may be blocked by another process. However, processes may access global information and, within slots containing components of Kind I, can allocate local memory. Of course, once a component of Kind I has started execution, the component cannot be halted by any external synchronization constraint, ensuring that it can proceed to release all memory that it has allocated.

While we have not exhaustively considered the types of programs that can be memory-limited, we hope that we have convinced the reader that many interesting and useful problems can be solved within the constraints of memory-limited programming.

5 MEMORY–EFFICIENT SCHEDULING

In order to ensure that a memory-limited parallel program delivers the desired performance, it is necessary that the actions of the program be scheduled in a suitable manner. The issue of scheduling is considered in this section.

A schedule can be defined in terms of mappings of actions to processors and to time intervals. However, a less formal approach will be sufficient for our purposes.

We will assume that all TAM systems are memory-limited.

**Definition.** (Completed, Delayed, Active, Ready) An action is completed if it has been executed by some processor. An action a is delayed if there is an action, b, such that b < a and b is not completed. An action is active if some processor has started executing the action but the action is not yet completed. Otherwise, the action is ready. A stint is completed if each action in the stint is completed. A stint is delayed if its start action is delayed. A stint is active if at least one action is active or completed but not all actions are completed. Otherwise, the stint is ready.

**Definition.** (Running Action) For each processor that is not idle, the action that the processor has most recently started to execute will be called the running action of the processor.

If a processor is between actions, then a running action as defined above may be completed. Otherwise, a running action is active. We will not allow actions to be preempted, so every active action will be the running action of some processor.

We can now define a memory-limited scheduling algorithm. (We will actually define a class of scheduling algorithms, since certain choices that are not important to our results are not fully specified.) For simplicity of presentation, we will assume that scheduling decisions are atomic and take no time. For example, two processors cannot simultaneously select the same action to run. Initially, all processors are idle.

**Memory-Limited Scheduling Algorithm**

- If a processor is idle and there is a least one ready start action, then the processor must start executing some ready start action, a. We will call _stint(a) the current stint of the processor._
- When a processor completes the last action of its current stint the processor becomes idle.
- When a processor completes an action, a, that is not the last action of its current stint, and _succ(a) is ready_, the processor must start executing _succ(a)._
- When a processor, p, completes an action, a, that is not the last action of its current stint, and b = _succ(a) is not ready, then it must be the case that b is an active or completed start action of a nested stint. Two subcases arise.
  - If _stint(b) is completed, then the processor takes the same action that it would have taken if it had just completed _max(stint(b))._
  - If _stint(b) is not completed, then it must be the current stint of another processor. In this case, the current stint of _p becomes the current stint of the other processor, and processor _p becomes idle. That is, the two stints are merged._

While for the most part, scheduling can be done locally on each processor, with some communication when _stints_ are merged, the problem of an idle processor finding a ready start action is not local. One solution, which is feasible with a limited number of processors and course grain parallelism, is to have a central pool of _stints_. Another approach to this problem is mentioned in Section 7.

**Definition.** (Current Initial Sequence) A current initial sequence is the initial sequence of the current stint of some processor up to and including the running action.

**Lemma 5.** Every active or completed action is in either a current initial sequence or a completed stint.

**Proof.** Clearly, the theorem is initially true, since there are no active or completed actions. The theorem is preserved by each of the transitions in the scheduling algorithm. Hence, by induction on the number of scheduling decisions, the theorem holds.
THEOREM 6. If $A$ is the set of actions in a memory-limited TAM system and the above scheduling algorithm is used with $m$ processors, then no more than $mM(A)$ units of memory will be allocated at any one time, where $M(A)$ is the maximum sequential memory requirement of the TAM system.

PROOF. The amount of memory allocated at any point in time will be

$$\Delta(t_{a} \mid a \text{ is completed or active}).$$

By Lemma 5, every active or completed action is in either a completed $sint$ or a current initial sequence. If an action is in a completed $sint$ that is contained in another completed $sint$ or current initial sequence, then only the largest completed $sint$ or current initial sequence will be considered, to avoid double counting of memory changes.

By the definition of memory-limited, the total amount of memory allocated by actions in completed $sints$ is at most zero. By Lemma 3, the total memory allocated by the actions in a single current initial sequence is at most $M(A)$. Since there are at most $m$ current initial sequences, the theorem holds.

Let us now consider the time efficiency of our scheduling algorithm. Any scheduling algorithm that allows a processor to be idle only if there is no work to do is called a work-conserving algorithm.

For any work-conserving scheduling algorithm, the worst possible schedule takes less than twice as long as the best possible schedule. In fact, the schedule length can be bounded in terms of the average parallelism in the problem (that is, the speedup that would be possible with an unbounded number of processors), and the number of processors. Let $r$ be the average parallelism, $T_1$ be the time required by the program on one processor and $T_m$ be the time required on $m$ processors, so $T_m = T_1 / T_m$ is the speedup on $m$ processors. Eager, Zahorian and Lazowska [4] have shown that $mr / (m + r - 1) \leq T_m < \min(r, m)$. It is always the case that $T_m > \min(r, m)/2$. If either $m$ or $r$ is very large compared to the other value, then the speed-up approaches the smaller of these two values. For example, if the average parallelism in a program exceeds the number of processors, then a speed-up equal to more than half the number of processors is guaranteed, and the guaranteed speed-up must approach the number of processors as the average parallelism becomes large. On the other hand, the speed up cannot be greater than the average parallelism or greater than the number of processors.

We should note, that an implementation of the scheduling algorithm needs to maintain only limited information about the structure of a TAM system as it unfolds at runtime. For example, a chain of actions will normally be treated as an indivisible unit for scheduling purposes. Serials may correspond to processes, and processes may be given a tree structure, with subprocesses corresponding to nested $sints$. It is necessary to be able to determine when a process becomes ready. In some cases, this suggested tree structure, a subprocess may be ready to execute before its parent process can start. However, if a process can determine the status of a child process (ready, active or completed) and communicate with an active child process, then it is fairly straightforward to determine what to do next in the case where a sequentially executing process comes to the point where, under sequential execution on a single processor, a child process would be started (i.e., the next action of a $sint$ is a start action, which may or may not have been executed already).

6 Nondeterministic Parallel Programs

Many parallel programs are nondeterministic. In some cases our results can be extended to nondeterministic programs.

We can view a nondeterministic program as a (possibly infinite) set of deterministic programs, with one deterministic program for each possible sequence of nondeterministic choices. If we can guarantee that each of these deterministic programs is a memory-limited program with good sequential memory performance, then we are guaranteed that the nondeterministic parallel program will have good parallel memory performance. This follows from the fact that the nondeterministic parallel execution will correspond to the execution of one of the deterministic programs.

Another approach is to view a nondeterministic parallel program as a nondeterministic sequential program. Often nondeterminism will result from the order in which work in a parallel program is performed. For example, the order in which processes read and update a shared variable may alter the behavior of a program. Unfortunately, understanding the behavior of such a program viewed as a nondeterministic sequential program may not be any easier than trying to understand the original parallel program. A more abstract approach to nondeterminism may solve this problem.

We may be able to identify all points where the memory requirements of a program may be altered by a decision that depends on nondeterministic values. Rather than consider all possible cases, we can assume that each decision will be made in an entirely arbitrary and independent manner. Since these decisions need not be independent, this may generate behaviors that would not actually be possible, but all possible behaviors of the program can still be generated. We now have a nondeterministic sequential program that can do anything that the original program could do, and perhaps more. The worst case memory requirements for this program must be at least as bad as those of the original program. For each decision we may be able to identify the outcome that will lead to the worst possible memory requirements. If we can modify the sequential program to always make the worst choice, and if the resulting program is memory-limited, then the sequential memory requirements of this deterministic program can be established and used as a basis for bounding the parallel memory requirements of the original program.

For example, consider a parallel combinatorial search program. Assume that a number of processes search different subspaces in parallel. These processes may decide whether to prune parts of the search space based on the values of global variables that are updated and accessed in parallel. Almost certainly, it will not be practical to consider all the possible sequences of values for these shared variables. However, as far as the memory requirements of the parallel program are concerned, the only decisions of
importance may be the decisions to prune or not. Let us assume that the parallel program will arbitrarily decide whether or not to prune without consulting the shared information. All other nondeterminism is irrelevant to the memory requirements of the parallel program, even though it may alter the result produced by the program. The arbitrary decisions may cause an incorrect result to be produced, but every correct execution of the program is still possible. If the worst case sequential memory requirements occur when pruning is never done, then we can modify the sequential program to never prune anything. We can now use this deterministic sequential program, if it is memory-limited, to determine an upper bound on the parallel memory requirements of the original program.

In general, we can transform a nondeterministic parallel program in any way that cannot improve the sequential memory requirements of the program. This always includes increasing the number of ways the program can behave, allocating additional memory, allocating memory earlier, and releasing memory later. If we can produce a memory-limited deterministic program that always requires at least as much memory as the transformed nondeterministic program, when both are executed sequentially, then the worst-case sequential memory requirements of the deterministic program can be used to bound the parallel memory requirements of the original program.

7 RELATED WORK

Many parallel programming languages sidestep the problems we have addressed, either by allowing only a fixed number of processes determined at compile time, or by not recovering the memory space of parallel processes [7].

However, a number of researchers have looked at the problem of bounding the memory requirements of parallel programs. Both theoretical and pragmatic approaches have been tried.

Early pragmatic approaches in the context of functional programming languages include [8]. Typically, functional programs contain far more implicit parallelism than a practical system can use. For example, with a simple parallel recursive program such as might arise using a balanced divide-and-conquer algorithm, a breadth-first parallel evaluation will cause the amount of parallelism to grow exponentially with respect to the depth of recursion. Unfortunately, the memory requirements also grow exponentially. To reduce memory requirements, processors locally execute work in the same order as it would have been executed with sequential computation. When a fork occurs, the processor continues working on the process that would have been executed first with sequential computation, but the other process becomes available for execution by another processor. If a processor needs work, it will attempt to steal the work from a neighboring processor. Simulation results indicate that this approach produces good performance, at least with the simple examples studied.

Later work with dataflow languages (essentially functional languages, but usually with “loops” and arrays) [9], [10] further develop methods for controlling parallelism, dynamically altering a systems behavior to keep the available parallelism sufficiently high while keeping memory requirements within acceptable bounds. These papers include some experimental results indicating the practical value of their approaches.

In a more procedural context, Halstead has designed a parallel version of Scheme called Multilisp [11]. Parallelism is explicitly generated by the programmer, using either of two constructs. One construct causes the arguments of a function to be evaluated in parallel. The other construct creates a future. In effect, the creation of a future informs the system that the result of a computation will be needed eventually and that it may start working on the computation immediately, but synchronization does not occur until the value to be computed by the future is actually required. Multilisp uses a LIFO queue (a stack) of processes, which in the case of a single processor would result in the ordinary sequential order of computation, and which in the case of multiple processors usually seems to keep memory requirements within acceptable bounds while providing sufficient parallelism.

Jagannathan and Philbin describe another parallel version of Scheme called STING [12]. However, their work is at a lower level. They state, “STING is intended to be used as an operating system substrate for modern programming languages.” Languages built on top of STING can control scheduling and the placement of work using the low level features of STING.

A more theoretical approach to bounding the memory requirements of the parallel programs that can be written in a language such as Language II from Section 4 has been taken by Burton in [5]. It is assumed that each program can be executed sequentially on a processor with a known amount of memory. For each process, the implementation keeps track of how much memory would have been available under sequential execution at the time the process would have started. In this way it is possible to ensure that once a process starts it will eventually have sufficient memory to finish. Unfortunately, no lower bound on the amount of parallelism is given. That is, the approach taken only ensures that a parallel program will not deadlock due to insufficient memory, and not that it will produce any particular speedup.

More recently, much more useful results have been given by Blumofe and Leiserson in [6], building on earlier results in [1]. Blumofe and Leiserson consider a class of programs that they call fully strict. (These include the class of programs that can be written in our Language II, and some other programs as well.) Basically, a fully strict parallel program consists of a tree of threads (or processes). A thread may allocate memory only when it is first started, and must release all the memory that it initially allocated at the time it terminates. This corresponds nicely to the allocation of stack frames in a recursive program. A thread may continue for a while after it has created a child thread, and may create more children during this time. However, for each child created by a thread, that thread must at some point halt and wait for the child to terminate if the child is still running. This corresponds to needing the result produced by the child. Only a parent may be delayed waiting for a thread to terminate. Blumofe and Leiserson also consider a more general class of programs called strict (or general strict) computations. In a strict computation, any ancestor of a thread, not just the parent, may be delayed waiting for the thread to terminate.
Both strict and fully strict programs are memory-limited, taking the sequential evaluation order to be depth first (e.g., run each child to termination immediately after it is first created). This is easy to see because any strict that contains the start of a thread will contain the entire thread.

Blumofe's and Leiserson's results are similar to ours, in the case of fully strict computations, in that they show that it is possible to ensure that any fully strict program that will run in n units of memory sequentially can run in mn units of memory on m processors, and the expected time required by the program is within a constant factor of the time required with an optimal schedule. Their expected time would be a worst-case time if they assumed, as we do, that an idle processor can find work in bounded time if any available work exists.

Their solution has a significant advantage over ours. They use a fully distributed scheduling algorithm. When a processor has no work to do it selects another processor at random and attempts to steal work from it, so no central pool of work is needed.

The advantage to our approach is that it applies to a larger class of programs. For example, Blumofe and Leiserson have not been able to use their approach with general strict programs, while our approach works with these and many other types of programs as well.

Bieloch, et al. [13], have developed a scheduling algorithm which guarantees a space bound of $S + O(pD)$, where $S$ is the space required for sequential execution, $p$ is the number of processors, and $D$ is the depth of the dag (i.e., the execution time on an unlimited number of processors, assuming unit execution time for each node). If $S$ is large relative to $D$, their approach will outperform ours. In some practical applications, $S$ may be large relative to even $pD$. In these cases, overall parallel memory requirements are $O(S)$ rather than $O(ps)$ as with our approach. On the other hand, if $D$ is large relative to $S$ (as may be the case with a program that requires a long time to execute, even with an unlimited number of processors, but never uses much memory at any given time), then our approach yields better performance. We should note that the approach used by Bieloch et al applies to a broader class of programs.

CONCLUSION

For a broad class of parallel programs, the scheduling algorithm given in Section 5 will ensure that a program will run with modest memory requirements while simultaneously ensuring asymptotically optimal speedup. Furthermore, an upper bound on the amount of memory required by a parallel program may be determined from a single experiment with a related sequential program.

The author, with others, is currently conducting further work related to that reported here. We are looking at other types of constraints that, together with suitable scheduling algorithms, are sufficient to ensure modest memory requirements and asymptotically optimal speedup for other types of parallel programs. We are looking at constraints that allow for more flexible use of memory and do not assume a unique sequential execution order, in exchange for a less tight restriction on parallel memory requirements. We are also considering memory management using reference counts or garbage collection, which makes the amount of memory released on an action nondeterministic in some cases, depending on whether other actions requiring an item of memory have been run. Our primary objective is the design of an efficient parallel functional programming language implementation. The results reported here are not sufficient for that purpose.

ACKNOWLEDGMENTS

The author would like to thank Bobby Blumofe, now at the University of Texas at Austin, David Simpson, at Simon Fraser University, and Vic Raymond-Smith, at the University of East Anglia, for helpful comments on earlier drafts of this paper. Blumofe found an error in one of the definitions, and suggested a way to simplify some of the results in the paper. The referees also made a number of helpful suggestions.

This work was supported by the Natural Science and Engineering Research Council of Canada.

REFERENCES

F. Warren Burton has a BS in applied mathematics, and an MA in mathematics from the University of Colorado, and a PhD in computing studies from the University of East Anglia.

Dr. Burton has been a professor of computing science at Simon Fraser University since 1988. Previously, he has served on the faculties of the University of Utah, the University of Colorado, the University of East Anglia, and Michigan Technological University. Dr. Burton is a member of the IFIP Working Group on functional Programming, and is on the editorial board of the Journal of Functional Programming.

His research interests include functional programming and parallel computing.