Partial Task Compaction Reduces Queuing Delays in Partitionable–Array Machines

O. Diessel *H. ElGindy* *B. Beresford-Smith*

Department of Computer Science and Software Engineering
The University of Newcastle
Callaghan NSW 2308
Australia

{*odiessel,hossam,bbs*}*@cs.newcastle.edu.au*

Abstract

Partitionable–array machines have emerged as popular target architectures for efforts to devise effective on–line processor allocation strategies. As the available processors become fragmented, contiguous processor allocation schemes can fail to allocate a task despite there being sufficient processors in total to service the request. Tasks consequently wait longer to be serviced, and the system response degrades. In this paper, we report on the use of partial task compaction to improve the performance of contiguous processor allocation methods for partitionable linear arrays. A quadratic sequential time algorithm to schedule the ordered compaction of tasks on a reconfigurable bus system is presented. The length of the schedule is shown to be within twice that of the minimum schedule length. Experimental results indicate that the algorithm almost eliminates the problem of fragmentation, and reduces the system response time by significantly reducing allocation delays.

1 Introduction

The partitionable multiple-SIMD/MIMD model allows compute processors to be shared among multiple, independently controlled tasks [9, 10, 3]. Such systems make effective use of the compute processors by adjusting the sizes of processor partitions to the sizes of the tasks, thereby allowing the number of tasks executed in parallel to be increased. Several contiguous processor allocation schemes have been proposed to manage the partitioning and allocation of contiguous blocks of processors under this model (see, for example, [6, 12]). Contiguous allocation schemes suffer from fragmentation of the available processors as variously sized tasks are allocated and deallocated. Tasks end up waiting in a queue to enter the system despite there being sufficient, albeit non-contiguous free processors available to service them. The time to complete a set of tasks is consequently longer, and the utilization of the compute resource is lower than it could be.

In this paper, we report on our investigation into the use of partial task compaction to reduce fragmentation in linear processor arrays. When a task is blocked from entering the system because the available processors are fragmented, we move a subset of the allocated tasks to combine the free fragments between them if doing so allows the task to enter the system sooner. In linear arrays the method almost eliminates the problem of fragmentation at the cost of introducing some execution delays due to task movements. Our partial task compaction algorithm reduces the allocation delays due to fragmentation as much as possible and attempts to minimize the execution delays due to task movements.

The use of full and partial task compaction methods to reduce fragmentation has previously been reported for hypercubes (see, for example, [2]) and meshes [11]. In this paper partial task compaction is investigated for a partitionable multiple–SIMD linear array of processors that, in addition to the local connections between neighbouring processors, is equipped with a reconfigurable bus to support rapid task compaction. We demonstrate how the allocation delay can be minimized by selecting a suitable compaction site. A heuristic compaction scheduling algorithm that allows a blocked task to be allocated in minimum time is developed. We also present results of an experiment to assess the effectiveness of our scheduling algorithm.

In the following section we describe the architecture and our assumptions in detail. We then describe our ordered partial task compaction method before presenting and analyzing algorithms to perform the task compaction efficiently in Section 3. Results on the use of task compaction to improve the performance of linear arrays augmented with reconfigurable buses that employ *first–fit* processor allocation and *first–come, first–served* scheduling are then reported in Section 4. Our findings are summarized in Section 5, which also mentions areas for future research.

2 Model

We consider a partitionable multi–SIMD linear array of compute processors, in which arbitrarily sized blocks of contiguous processors are independently controlled to operate in SIMD mode. An overview of this model is given in Figure 1. The compute resource consists of n interconnected processing elements (PEs) that are controlled by a set of m control processors (CPs) under the global control of a host. The host orchestrates the operation of the CPs, each of which broadcasts instructions for the PEs under its control over the CP–PE interconnection network.

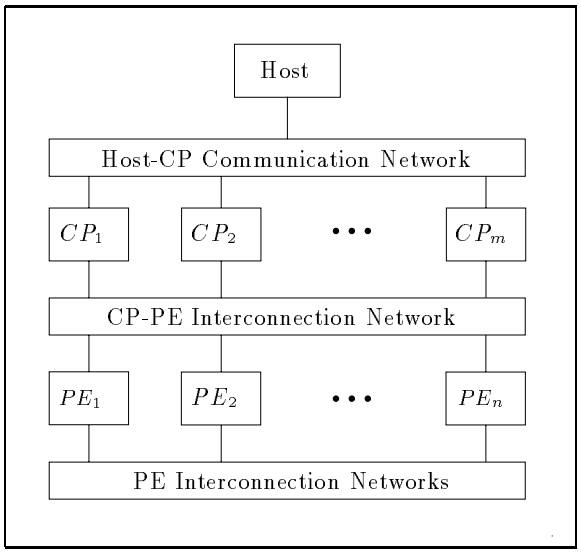

Figure 1: The multi–SIMD model of parallel computation.

In our model, the PEs are connected to their neighbours to the east and west for intra–task communications. In addition, PEs are connected to the base of a reconfigurable mesh of switching elements to the north, as illustrated in Figure 2, for the purpose of task movements.

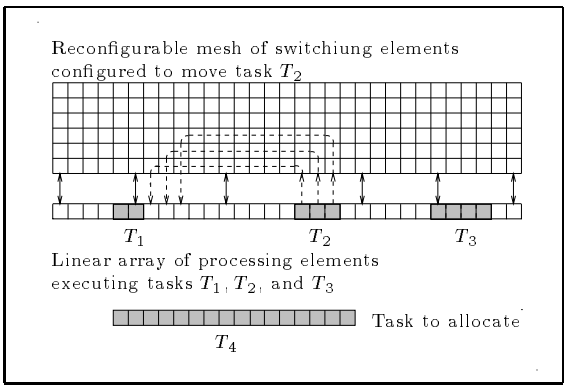

Figure 2: Problem model depicting the move of T_2 to accommodate T_4.

A reconfigurable mesh [7] of size $m \times n$, consists of m rows and n columns of processing elements arranged in a grid. Traditionally, these processors are fully–fledged SIMD processors. However, for this application, a simple sequential circuit capable of the comparisons and address calculations needed to set the reconfigurable mesh switches will suffice. For the remainder of this paper we refer to such a circuit as a switching element (SE). Each SE is connected to its immediate neighbours to the north, south, east and west, when present, and has four similarly labeled I/O ports through which it can communicate with its neighbours. Each SE has control over a local set of short–circuit switches that allow the four I/O ports to be connected together in any combination.

The SEs operate synchronously, in one machine cycle performing an arithmetic, logic or control operation, setting a connection configuration, and sending (receiving) a datum to (from) each I/O port. When a connection is set, signals received by a port are simultaneously available to any port connected to it. For example, if the SEs connect their northern and southern I/O ports by closing the appropriate switches, data "broadcast" onto the "column bus" thus formed can be read by all of the processors in a column. The model allows concurrent reading from a bus, requires exclusive writing to a bus, and assumes a constant time communication delay on arbitrarily long connected bus components. A similar reconfigurable mesh of switching elements can be used to implement the CP–PE interconnection network.

Moving a task involves switching the task out of context and setting the mesh to form buses to move the task elements (processor contexts and local memories). The source (target) processors of a move then write (read) their task elements to (from) their mesh ports. When all the elements of a task have reached their destination, the task is switched back into context. Compute processors have storage for two task elements between which they can switch in a single step. Storage for a task element that is switched out of context may be accessed from the compute processor's mesh port in a single step while the processor participates in the execution of another task. Compute processors participate in the execution of at most one task at a time, and all of the elements of a task need to be switched in for the task to be able to execute. Compute processors can participate in the execution of a task at the same time as communicating with the mesh. Forming a bus within the reconfigurable mesh, and writing (reading) a task element to (from) context storage can be done in a single step. However, several bus–forming and broadcasting phases may be needed to compact tasks unless the height of the mesh, H, is equal to its width, W. If $H = W$, then it is straightforward to see that compaction can be accomplished in $O(1)$ steps.

Overall management of tasks is accomplished in the following way: Tasks are queued by a sequential host as they arrive. A task allocator, executing on the host, attempts to find a location for the next pending task. If some executing tasks need to be compacted to accommodate the next pending task, then a schedule for switching those tasks out of context, and moving their task elements over the mesh is computed by the allocator. The allocator coordinates the switching of the reconfigurable mesh, task processors and task controllers according to the compaction schedule, and associates an available control processor with the new task and its allocation. If a location for the next pending task cannot be found, the task waits until one becomes available following one or more deallocations as tasks complete processing.

3 Algorithms

In this section we address the problem of allocating the task at the head of the queue (henceforth called the *incoming task*) as quickly as possible when no contiguous block of free processors is large enough to satisfy the request although sufficient free processors in total are available. The task compaction process consists of: selecting a suitable location for the incoming task, moving occupying tasks to new locations, and allocating the new task.

Task compaction methods in linear arrays may be classified according to whether they preserve or permute the spatial order of tasks, and whether they move tasks in one or two directions. In this paper, we focus on order–preserving compaction of tasks, and discuss the one–way case in detail. We derive a lower bound on the time needed to move a task and use it to select a location for the incoming task that minimizes the time needed to allocate it. We then describe a scheduling algorithm to sequence the task moves so as to free the space needed by the incoming task in minimum time.

3.1 Left Ordered Compaction

The idea behind Left Ordered Compaction is to choose a suitable location for the incoming task, and to move the tasks already occupying the processors at that location to the left while preserving their spatial order. In general, these tasks will be moved to locations occupied by other tasks that will need to be moved as well. Our Left Ordered Compaction protocol identifies a sequence of tasks that needs minimal time to be compacted, and computes a schedule for compacting the tasks so that they abut one another and vacate sufficient processors on their right for the incoming task.

3.2 Selecting an optimal location for the incoming task

In order to minimize the time to allocate the task, we choose a location for the incoming task that takes minimum time to free of executing tasks. The following lemma describes the criterion for computing the optimal time required to move a task. This information allows us to compute the ordered compaction schedule length for a subset of tasks, and thus to choose a location that requires minimum time to free of executing tasks. To simplify the presentation, we assume all lengths and distances are multiples of the reconfigurable mesh height, H.

Lemma 1 *[4] A task of length l can be moved d processors to the left in $\Theta(\min(\frac{l}{H}, \frac{d}{H}))$ steps using a reconfigurable mesh of height H. This is optimal.*

Let us say a task T_i is *covered* by another task T_j, if the processors allocated to T_i would need to be reallocated to T_j were T_j to be based at a particular location. From Lemma 1, the time (cost) to move a task T completely covered by the incoming task is proportional to the length of T, whereas the cost to move a task T partially covered by the incoming task is dependent on the direction it is moved in: if the uncovered portion of T is moved away from the incoming task, then the cost to move it is proportional to the number of processors covered, otherwise, the cost is proportional to the length of T. Let each executing task T_i be associated with a free block f_i to its left, possibly of zero length. We say the tasks $T_i, T_{i+1}, \ldots, T_j, i \leq j$ are a *contiguously allocated sequence* of tasks if the free blocks $f_{i+1}, f_{i+2}, \ldots, f_j$ between them all have zero length. The sequence is a *maximal contiguously allocated sequence* if the free blocks f_i to the left of T_i, and f_{j+1} to the right of T_j, have non–zero length. The following theorems allow us to identify an optimal location for the incoming task efficiently.

Theorem 1 *[4] The time needed to free processors for the incoming task using Left Ordered Compaction is minimized when the rightmost task element of the incoming task is allocated to the rightmost processor of a free block of non–zero size.*

Theorem 2 *[4] It is necessary to consider both Left, and the symmetric Right Ordered Compaction of tasks in order to minimize the time needed to free the processors for the incoming task using an Ordered Compaction approach.*

Theorem 1 allows us to only consider the rightmost processors of each non–zero sized free block as possible locations for aligning a window as wide as the incoming task. For each such location we compute the time to move the allocated tasks, which is

equal to the total number of allocated processors within the window. The location that minimizes the time, and that has sufficient free processors to the left to accommodate the allocated processors within the window, is the best location for the incoming task using Left Ordered Compaction. Theorem 2 asserts that the times required by Left and Right Ordered Compaction to free the processors for the incoming task are not necessarily equal, so both need to be considered to minimize the compaction time. The optimal location can be found in linear sequential time by scanning a list of task and free block records in base processor order as outlined in Procedure **SelectTasksToCompactLeft**. A list of tasks that need to be moved, together with their new base locations, can be compiled during the scan.

Procedure **SelectTasksToCompactLeft**

Input Doubly linked list of task and free block records containing the size of records in base processor order. The size of the request.

Output Pointers to the rightmost and leftmost tasks in the sequence of tasks to compact.

begin

1. scan forwards through the list with pointer R until R points to a non–zero sized free block record and the total free space to the left of $R.next$ exceeds the request

2. repeat

   (a) if the incoming task location that allocates the rightmost task element to the rightmost processor of R covers the least allocated processors so far then

      i. scan forwards through the list with pointer L until the free space between $R.next$ and $L.next$ is less than the size of the request

      ii. save pointers to the rightmost ($R.prev$) and leftmost ($L.next$) tasks in the sequence to compact

   (b) scan forwards through the list with R until R points to a non–zero sized free block record or the list has been scanned

   until the list has been scanned

3. return saved pointers

end

3.3 Scheduling the task moves

A schedule to compact the tasks is needed when the height of the reconfigurable mesh is less than the number of task elements that are to be moved. Our scheduling goals are:

1. to move the tasks occupying processors that are to be allocated to the incoming task as quickly as possible,

2. to avoid delaying the tasks that are to be compacted any more than necessary to move them, and

3. to complete compacting the tasks as soon as possible.

In the remainder of this section, we motivate the need for a schedule and specify the requirements of a partial task compaction schedule. We obtain a lower bound on the schedule length, and outline our strategy for minimizing delays to tasks. We then present an algorithm that allocates the incoming task in minimal time, and obtain bounds on the maximum delay to executing tasks and the maximum schedule length.

For a sequence of tasks at given locations and a given reconfigurable mesh height, a *Left Ordered partial task compaction schedule* defines for each task element that is to be moved, the step of the schedule it is to be switched out of context in, the step it is to be moved in, the step it is to be switched back into context in, and designates the row of the mesh to be used for the transfer.

Definition 1 *We use the symbol S_O to denote the number of blocks of H task elements covered by the incoming task.*

Lemma 2 *[4] Any Left Ordered partial task compaction schedule will require at least S_O or $S_O + 1$ steps to complete, depending on whether the block of H processors immediately to the left of the incoming task location is free or not, respectively.*

In general, more than $S_O \times H$ task elements in total will need to be moved, since the occupying tasks are moved on top of other executing tasks. Several tasks therefore need to be moved simultaneously, and in each step several task elements will need to share a row of the reconfigurable mesh in order for the compaction schedule to meet the lower bound.

The tasks occupying processors that are to be allocated to the incoming task need to be switched out of context or moved in order to commence executing the incoming task. If these tasks are not moved as soon as they are switched out, they are delayed from executing, which is to be avoided since delays increase response times. Tasks occupying the processors that are to be allocated

to the incoming task therefore need to be moved before the incoming task can commence executing. To avoid delaying executing tasks any more than necessary, they are moved just once as soon as they are switched out of context, the task elements are moved in a minimum number of consecutive steps, and tasks are switched into context again as soon as they complete their moves. Whether or not it is always possible to move the tasks occupying processors that are to be allocated to the incoming task in minimum time without delaying any executing tasks more than the minimum time necessary to move them is still under investigation.

Our ordered compaction schedule is obtained by identifying sequences of task elements that can be moved in parallel. The sequences are defined so that the task elements within each sequence can be moved without affecting the moves of task elements in neighbouring sequences. Task elements within a sequence commence moving with the rightmost element in the first step of the schedule, and proceed to move in decreasing address order until the leftmost element has been moved. If the rightmost sequence does not include all of the task elements that occupy processors that are to be allocated to the incoming task, then it does not commence moving until its rightmost element is unimpeded by movements in the first sequence to its left. The schedule is complete when the leftmost task elements of all sequences have been moved. Figure 3 illustrates a schedule for a Left Ordered Compaction instance assuming $H = 1$. Sequence 1 covers the 6 task elements at processors 21,...,24, 29, and 30. These can be moved in parallel with those of sequence 2 (15, 17,...,19). The rightmost sequence does not include all the allocated processors covered by the incoming task (24, 29,...,33), hence the rightmost task element of the rightmost sequence is impeded from moving until the fifth step of the schedule.

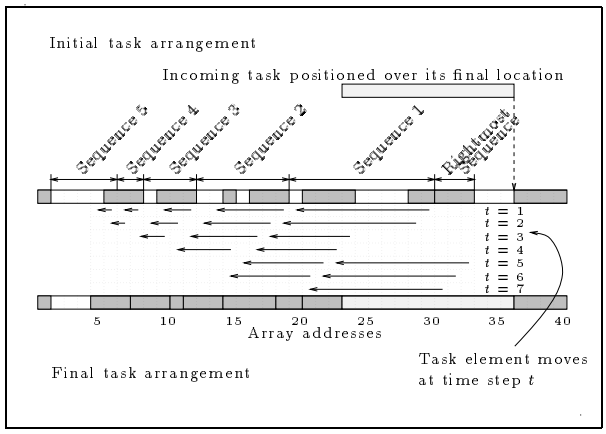

Figure 3: A Left Ordered Compaction schedule.

We compute a *candidate schedule* commencing with the rightmost task element of each task that occupies processors that are to be allocated to the incoming task. From among these candidates, we choose the schedule that minimizes the delay to moved tasks, and if there are two or more such schedules, then one that minimizes the schedule length. To complete this section, we describe in detail how the sequences for a candidate schedule are found, give bounds on the schedule length and delay caused to executing tasks, and derive the time complexity of a sequential algorithm to find the best schedule.

Let us assume a sequence of tasks has been selected for Left Ordered Compaction. For each task that occupies processors that are to be allocated to the incoming task, the sequences are identified by finding those task elements that can be safely moved in parallel with the rightmost task element. These form the rightmost task elements of the sequences for a candidate schedule. As described below, the candidate schedule length, and the maximum delay to a task can be computed during the sequencing operation, thus the schedule that minimizes the maximum delay to executing tasks and the total schedule length can be chosen once all candidate sequences have been computed. Say task T_i of length l_i, with free block f_i (possibly of zero size) to its left, is to be moved a distance d_i to the left. We refer to the task elements of T_i as $T_i(j), 1 \leq j \leq l_i$.

Suppose the sequence of tasks to be moved is $T_r, T_{r+1}, \ldots, T_{r+s}$, and that some task element $T_j(v)$, $r \leq j \leq r+s, 1 \leq v \leq l_j$ has been chosen to be moved in the first step. We consider where $T_j(v)$ is moved to to find the next task element to the left that can be moved in parallel with it on the same row of the reconfigurable mesh. To simplify the discussion, we assume $H = 1$. There are three cases:

1. If $T_j(v)$ is moved to a free processor of some free block f_i, then $T_{i-1}(l_{i-1})$, if it exists and needs to be moved, can be moved to the left together with $T_j(v)$.

2. Should $T_j(v)$ be moved to a processor occupied by $T_i(u), i \leq j, 1 \leq u \leq l_i$, of a task T_i with $d_i \leq l_i$, then no task element from tasks to the right of T_i are moved to the left of T_i since the processor abutting T_i on the left will be occupied by a task element of T_i. If $u > 1$, then the processors allocated to the task element $T_i(u-1)$, and those to its left, can write to the left. At the same time, the processors allocated to $T_i(u)$, and the task elements to its left, read from the right. After $T_{i+1}(1)$ has moved, the task elements $T_i(l_i), \ldots, T_i(u)$ are moved by the sequence commencing with $T_j(v)$. If $u = 1$, then $T_{i-1}(l_{i-1})$, if it exists and needs to be moved, can be moved to the left together with $T_j(v)$.

3. When $T_j(v)$ is moved to a processor occupied by $T_i(u), i < j, 1 \leq u \leq l_i$, of a task T_i with $d_i > l_i$, then some task element from tasks to the right of T_i is moved to the left of T_i since the processor abutting T_i on the left will not be occupied by a task element of T_i. Since the task elements of T_i to the right of, and including $T_i(u)$ cannot move while the tasks $T_{i+1}, \ldots, T_j$ are moved to the left of $T_i(u)$, T_i must wait until after they have moved for it to be able to move in the least possible number of consecutive steps. In this case, $T_{i-1}(l_{i-1})$, if it exists and needs to be moved, can be moved in parallel with $T_j(v)$.

The above selection is repeated for the task element just found until it is not possible to choose another task element to the left that can be moved in parallel with $T_j(v)$. The identified task elements form the rightmost task elements of the sequences for a candidate schedule commencing with $T_j(v) = T_j(l_j)$ for some task $T_j, r \leq j \leq r+s$ occupying processors to be allocated to the incoming task. If $T_j(l_j) \neq T_{r+s}(l_{r+s})$, then the rightmost sequence commences with $T_{r+s}(l_{r+s})$ after $S_O - \sum_{i=j+1}^{r+s} l_i$ steps if the processor to the left of the incoming task location is free, or one step later, if not.

Lemma 3 *[4] The distance moved by a task T_i whose leftmost task element is not covered by the incoming task is at most S_O processors.*

Lemma 4 *[4] The number of allocated processors spanned by a bus used to move a task element is at most $S_O + 1$.*

The number of steps needed to complete a candidate schedule, S_L, is given by the maximum number of task elements spanned by any sequence in the schedule. The rightmost sequence completes after S_O steps if the processor to the left of the incoming task is free, or after one more step, if not.

Lemma 5 *[4] A candidate schedule length S_L is at most $2S_O$.*

An executing task is *delayed* if it is switched out of context for more steps than the minimum number needed to move it. The delay to an executing task is the number of steps it is switched out and none of its elements is moving.

Lemma 6 *[4] The maximum delay to an executing task is at most $S_O - 1$ steps.*

Theorem 3 *[4] For Left Ordered compaction of tasks, there is a schedule with length at most $2S_O$ that delays no task more than $S_O - 1$ steps. Moreover, such a schedule can be found in quadratic sequential time.*

Corollary 1 *[4] Two-way ordered compaction has the potential of reducing the maximum schedule length and the maximum delay to executing tasks due to Left Ordered compaction by a factor of two.*

Procedures **ComputeLeftOrderedSequences** and **ComputeBestLeftOrderedSchedule** contain an outline of the algorithm.

Procedure **ComputeLeftOrderedSequences**

Input Doubly linked list of task records $T_r, T_{r+1}, \ldots, T_{r+s}$ containing the size of each task and the distance it is to be moved in base processor order. Pointer to a task T_k whose rightmost task element $T_k(l_k)$ is to be moved in the first step.

Output A list of task elements to be moved in parallel with $T_k(l_k)$ in decreasing address order. The maximum delay $\delta_{\max}$ to any task. The candidate schedule length S_L

begin

1. set $T_k(l_k)$ to be the task element $T_j(v)$ to be moved in the first step
2. while a task element $T_j(v)$ remains to be moved
   (a) append the task element to the list of task elements to be moved in the first step
   (b) find the next task element to the left of $T_j(v)$ that can be safely moved in parallel with it
   (c) if $T_j(v)$ is moved to T_i with $l_i < d_i$ then compute and save the maximum delay
   (d) compute and save the length of the sequence commencing with $T_j(v)$
   (e) set $T_j(v)$ to be the next task element to the left to be moved

end

Procedure **ComputeBestLeftOrderedSchedule**

Input Doubly linked list of task records $T_r, T_{r+1}, \ldots, T_{r+s}$ containing the size of each task and the distance it is to be moved in base processor order. The size of the request.

Output A schedule specifying for each time step the addresses of task elements that are to be switched in and out of context, and the source and destination addresses of task elements that are to be moved.

begin

1. for each task T_k whose rightmost processor is to be allocated to the incoming task

(a) call **ComputeLeftOrderedSequences** with $T_r, \ldots, T_{r+s}$, and pointer to T_k, returning the list of task elements to be moved in the first step of a schedule commencing with $T_k(l_k)$, the maximum delay to tasks $\delta_{\max}$, and the candidate schedule length S_L

(b) if ($\delta_{\max} < \delta_{\min}$ or ($\delta_{\max} = \delta_{\min}$ and $S_L < S_{L_{\min}}$)) then save the list of task elements, $\delta_{\min}$ and $S_{L_{\min}}$

2. derive the schedule for the sequences defined by the saved list of task elements

end

4 Experimental Results

We conducted a series of experiments to evaluate the benefits of task compaction and to assess the effectiveness of the Left Ordered Compaction method. Requests for service consisting of task sizes and service times were derived from trace data obtained from a 400 node Intel Paragon at San Diego Supercomputer Center (SDSC) [1] and a 128 node iPSC/860 at Numerical Aerodynamic Simulation (NAS) Systems Division, NASA Ames Research Center [8]. These were presented to a linear array simulator in their original order using uniformly distributed random intertask arrival periods. We examined the performance of the simulated system as the maximum intertask arrival period was varied using three allocation methods:

1. First-fit (FF) - a task was allocated to the leftmost free block of contiguous processors large enough to satisfy the request.

2. Left Ordered Compaction (LOC) - used the Left Ordered Compaction method to allocate a task when the total number of free processors exceeded the request and the request could not be satisfied by FF.

3. Cost Free Compaction (CFC) - executing tasks were compacted *instantaneously* when the required number of free processors were available, but not contiguously. Scheduling and compaction costs were not accounted for. CFC used FF when compaction was not necessary.

Our results are summarized in the following sections. A more detailed description of the experimental method and results is available from [4].

4.1 SDSC Trace Results

When tasks arrive more rapidly than they can be processed, the system is said to be *saturated*. Under these conditions, the average amount of time a request spends advancing to the head of the pending queue, the *average queue delay*, is proportional to the difference in the rate at which tasks arrive, and the rate at which they are allocated. We observed a reduction in average queue delay from FF to LOC (CFC) of 17.2% (17.5%) when all requests were assumed to have arrived at the same instant. This improvement increased as the average intertask arrival period increased to levels at which the system did not saturate. The closeness of the results for LOC and CFC indicates that the costs associated with compaction are insignificant. The reduction in average queue delay was a result of a reduction in average allocation delay due to compaction.

The *average allocation delay* is the amount of time the request at the head of the pending queue waits to commence processing on average. This includes the time the request waits for sufficient processors (for the allocation method) to become available, as well as the time to allocate those processors and to load the incoming task. We observed a reduction in average allocation delay from FF to LOC (CFC) of 16.5% (16.8%) in the saturated system. See Figure 4. The average allocation delay, and the improvement due to compaction, were constant until the maximum intertask arrival period exceeded the average allocation delay at saturation. The reduction in average allocation delay with compacting allocation methods is due to their ability to combine the free processors in order to satisfy the request at the head of the queue. The amount of reduction at saturation is dependent on the size of the system, and the distribution of task sizes and task service times. The compacting allocation methods were also observed to sustain higher intertask arrival rates before saturating due to their lower average allocation delay at saturation.

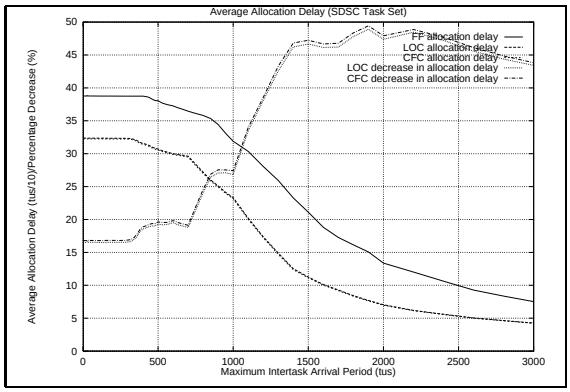

Figure 4: Average allocation delay to SDSC tasks arriving at uniformly random intervals on a linear array of 400 nodes using First–fit, Left Ordered, and Cost Free Compaction allocation methods.

For the LOC (CFC) allocation method, we observed a 16.3% (16.5%) reduction from FF in the

amount of time needed to complete processing the task set (*completion time*) at saturation, and we also obtained a 19.4% (19.8%) increase in system utilization due to LOC (CFC). Benefits of this size were to be expected, since at saturation, completion time is proportional to the average allocation delay, and utilization is inversely proportional to the completion time.

As the average intertask arrival period rose above the average allocation delay at saturation, the average queue and allocation delays were observed to fall towards zero. The completion times were observed to increase linearly, and the utilizations to decrease inversely with further increases in average intertask arrival period. While average queue and allocation delays were still significantly less for LOC and CFC than for FF at high intertask arrival periods, there was no reduction in completion time or increase in utilization due to compaction when the system was no longer saturated because these performance measures then depend upon the rate at which tasks arrive rather than the rate at which they are allocated.

The delay to tasks resulting from moving them was investigated for individual tasks in the SDSC task set at saturation when the number of allocations using compaction was highest. The average results for 10 runs are recorded in Table 1.

	Tasks finishing later with LOC	Tasks finishing at the same time	Tasks finishing earlier with LOC
Num tasks	10.0	82.0	32,966.0
Max % incr (decr) finish time	0.9	0.0	(96.6)
Mean % incr (decr) finish time	0.2	0.0	(19.4)
Num tasks delayed	10.0	0.0	4,765.2
Max % incr run time	1.0	0.0	48.7
Mean % incr run time	0.2	0.0	0.8

Table 1: Delays to SDSC tasks due to task movements at saturation using Left Ordered Compaction, compared with times for the same tasks allocated using First Fit.

Although 14.4% of tasks experienced extended running times as a result of being moved, only 0.03% of tasks finished later than they would have with FF allocation. These were tasks that arrived early enough for them to be allocated at the same time under both FF and LOC, but that were moved at some time due to their extremely long execution times. The overwhelming majority of tasks finished earlier with LOC, even when their running times were extended. Running times were extended on average by less than 1%.

4.2 NAS Trace Results

When the NAS task set was processed on a 128 node array we obtained a similarly shaped response in the key performance indicators, however, the magnitude of the benefits due to compaction were much lower. A reason for the depressed performance of compaction was found in the distribution of task sizes relative to the size of the array. The NAS task set contained relatively more large requests than the SDSC task set did. Since the performance of compaction appeared to depend quite dramatically on the task sizes relative to the array size, we examined the relationship between performance and array size at saturation more closely using the NAS task set. The results appear plotted in Figure 5.

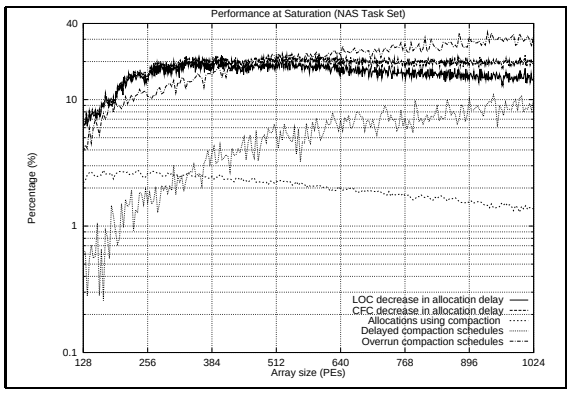

Figure 5: Dependence of LOC performance on array size when the NAS task set was processed with a maximum intertask arrival period of 2 time units.

The reduction in average allocation delay due to compaction increased to approximately 20% when the array was about three times as large as the largest request. This observation is supported by Knuth's finding regarding segmented memory management, that requests for more than one third of memory would be refused with high probability [5]. The performance gap between LOC and CFC increased and the number of allocations using LOC compaction decreased with increasing array size. We think the higher degree of parallelism on larger arrays increased the cost of LOC relative to CFC due to the increase in the cost of selecting a compaction location. The number of allocations using LOC would have decreased with higher levels of parallelism due to the increased likelihood that suitable tasks would have been deallocated during the scheduling phase, thereby causing LOC to abort compaction more frequently. We attribute

the increase in the number of delayed and over–running compaction schedules to increasing parallelism as well, since it would have made it more likely that tasks would be moved to processors occupied by other tasks.

5 Concluding Remarks

Partial task compaction reduces the allocation delay to tasks, which in turn reduces the queue delay and the time to complete a given task set and increases the utilization of the compute resource. These benefits are greatest when tasks arrive more frequently than they can be processed, although significantly reduced allocation and queue delays can be expected even when tasks arrive infrequently. It was found that allocating tasks with compaction increases the load that can be sustained by the system before it saturates. The load–bearing capacity increased in proportion to the reduction in the average allocation delay. The benefits of compaction vary according to the task mix, and appear to be more significant when the sizes of the tasks are small compared to the system size. Performance benefits of over 15% were obtained for task sets derived from actual trace data assuming tasks were waiting to be processed. Under these conditions, the overwhelming majority of tasks finished earlier when realistic compaction costs were included. The reduced finishing times resulted from significantly reduced queue delays, which were offset by slight increases in processing times.

Partial task compaction reduces the allocation delay to linear SIMD tasks by alleviating the problem of fragmented idle processors. While the problem cannot be eliminated entirely, since there is some cost involved in scheduling and carrying out task movements, simulations of the simple Left Ordered Compaction method performed almost as well as if these costs did not exist. Although our Left Ordered Compaction scheduling algorithm is not optimal, it appears to be practical. The problem of deciding whether a Left Ordered Compaction instance can be scheduled in the minimum time needed to free the incoming task location of occupying tasks without delaying executing tasks remains open.

References

[1] S. D. S. Center. Intel Paragon trace data. Available by ftp from `ftp.cs.uoregon.edu/pub/lo/trace/sdsc.tar.gz`.

[2] H.-L. Chen and N.-F. Tzeng. Task migration in hypercubes using all disjoint paths. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Sy stems*, pages 358–363, Oct. 1994.

[3] Cray Research, Inc. *Cray T3D system architecture overview*, Sept. 1993.

[4] O. Diessel, H. ElGindy, and B. Beresford-Smith. Partial task compaction reduces queuing delays in partitionable–array machines. Technical report 96-06, Department of Computer Science and Software Engineering, The University of Newcastle, 1996. Available by anonymous ftp: `ftp.cs.newcastle.edu.au/pub/techreports/tr96-06.ps.Z`.

[5] D. E. Knuth. *The Art of Computer Programming Vol 1, Fundamental Algorithms*. Addison–Wesley, Reading, Massachusetts, second edition, 1973.

[6] K. Li and K.-H. Cheng. A two–dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. *Journal of Parallel and Distributed Computing*, 12(1):79–83, May 1991.

[7] R. Miller, V. K. Prasanna-Kumar, D. I. Reisis, and Q. F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42(6):678–692, June 1993. (A preliminary version of this paper was presented at *5th MIT Conference on Advanced Research in VLSI*, 1988).

[8] B. Nitzberg. NAS iPC/860 Workload Data 4Q93. Personal communication. Available by ftp from `ftp.cs.uoregon.edu/pub/lo/trace/nas.tar.gz`, July 1994.

[9] H. J. Siegel, L. J. Siegel, F. C. Kemmerer, P. T. Mueller Jr., H. E. Smalley Jr., and S. D. Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C–30(12):934–947, Dec. 1981.

[10] Supercomputer Systems Division, Intel Corporation, Beaverton, OR. *Paragon XP/S Product Overview*, 1991.

[11] H.-y. Youn, S.-M. Yoo, and B. Shirazi. Task relocation for two–dimensional meshes. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 230–235, Oct. 1994.

[12] Y. Zhu. Efficient processor allocation strategies for mesh–connected parallel computers. *Journal of Parallel and Distributed Computing*, 16(4):328–337, Dec. 1992.