Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 33 (2007) 145–158
PARALLEL COMPUTING
www.elsevier.com/locate/parco

Distributed dynamic processor allocation for multicomputers

César A.F. De Rose [a], Hans-Ulrich Heiss [b,*], Barry Linnert [b]

[a] *Catholic University of Rio Grande do Sul (PUCRS), Department of Computer Science, Av. Ipiranga, 6681 – Building 32, 90619-900 Porto Alegre, Brazil*

[b] *Technical University Berlin, Faculty of Electrical Engineering and Computer Science, Einsteinufer 17, Sec. EN 6, 10587 Berlin, Germany*

Received 5 June 2006; accepted 23 November 2006
Available online 20 December 2006

Abstract

Current processor allocation techniques for highly parallel systems use centralized front-end based algorithms which restrict applied strategies to static allocation, low parallelism, and weak fault tolerance. To lift these restrictions, we are investigating a distributed approach to processor allocation in multicomputers where currently no centralized data structure with information about the state of all processors exists. This approach will allow the implementation of more complex allocation schemes and possibly the consideration of dynamic allocation, where parallel applications would be able to adapt the allocated processor partition to its actual demand at running time, resulting in a more efficient utilization of system resources. Noncontiguous versions of a distributed dynamic processor allocation scheme are proposed and studied in this paper as an alternative for parallel programming models to allow dynamic creation and task deletion. Simulations compare the performance of the proposed dynamic strategies with static counterparts and also with well-known centralized algorithms in an environment with growing and shrinking processor demands. To demonstrate dynamic allocation is feasible with current technologies, results of the experiments are presented for a 96 nodes SCI hpcLine Primergy Server cluster.

Keywords: Resource management; Scheduling; Dynamic processor allocation; Distributed algorithms; Multicomputer

1. Introduction

Parallel machines with distributed memory, such as massively parallel processing systems (MPP) or cluster computers are called *multicomputers*. Their processing nodes consist of a processor and private memory and are connected by a network in order to exchange messages. Jobs to be run on those systems are parallel programs consisting of tasks which communicate with each other. We assume that upon arrival, each program requests a specific number of processing nodes large enough to accommodate all tasks. Such a request is usually satisfied by allocating a sufficiently large subset of processors to the program. The selection of that subset in order to maximize some performance measure is a particular instance of a *resource management problem*

* Corresponding author. Tel.: +49 30 314 73161; fax: +49 30 314 25156.
E-mail address: heiss@cs.tu-berlin.de (H.-U. Heiss).

0167-8191/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2006.11.010

and referred to as *processor allocation problem*. The selection of a job from the input queue (scheduling) is not considered here. For the rest of the paper, first come first served (FCFS) is implicitly assumed. Improvements over FCFS such as different variations of backfilling would also be applicable here, but are not in the scope of this paper. Once a set of processors has been allocated to a program, a second allocation problem must be addressed: which particular task of the parallel program is assigned to which processor. This is called the *mapping problem* and is usually represented as a graph-matching or graph-embedding problem since both the communicating tasks and the processor network can be modeled as graphs. This paper addresses the problem of selecting an appropriate processor subset to a job, a *partition*. It is implied that resulting partitions are disjointed.

1.1. Processor allocation

Processor allocation must meet several partly contradicting goals:

1. *High utilization*
   Processor allocation must maximize resource utilization, i.e., it must avoid any kind of fragmentation so that all processors can be used.
2. *Low overhead*
   Since all requests are processed at run-time, resource allocation algorithms must be fast and cause only low overhead.
3. *Scalability*
   Algorithms must support systems of thousands of nodes without creating a bottleneck.
4. *Low latency*
   Low execution times of parallel programs must be supported. In some machines, execution time will be affected by the allocation scheme with regard to communication bandwidth and latencies within the partition. Although nothing may be known about communication patterns of parallel programs that occupy those partitions, it is assumed that arbitrary tasks of the program communicate with each other. Partitions with low diameters and a large number of internal links generally lead to better communication performance, e.g., in a 2D-mesh, a partition in the form of a square would better serve an arbitrary program than one in the form of a narrow and long strip. This can be reconsidered in machines where node distances do not significantly affect message latency.

Despite some specific applications where a program is running permanently on a dedicated machine, it is almost inevitable in large systems with 100 or 1000 of nodes, to allow *multiprogramming*, i.e., several parallel programs sharing the machine in order to achieve high machine utilization. Two ways of sharing exist: *space sharing* and *time sharing*. Space sharing is when each program is exclusively given its own set of processors.

Although there is theoretical and empirical evidence that time sharing can significantly boost the utilization of the processors [1], in this context it is not yet widely used. One reason may be that a large amount of memory per node is needed to accommodate all programs and avoid paging. Since allocation operations must be executed efficiently at load time, ordinary allocation techniques restrict feasible shapes of partitions to achieve some regularity, and facilitate management. A partitioning scheme can be called *structure preserving* if it generates partitions that are of the same topological graph family as the entire processor graph, specifically, subcube allocation in hypercubes and submesh allocation in meshes. Because task interaction has to cross-node boundaries, communicating tasks should be placed closely together to maintain low communication delays. In most cases, *contiguous* partitions are useful, meaning their processors are constrained to remain physically adjacent. For example, in a 2D-mesh, each request would be served by exactly one rectangular partition of sufficient size.

The direct consequence of such simplifications in the allocation scheme is that a 100% machine utilization is not achievable due to two types of *fragmentation*: *internal fragmentation*, when processors are allocated, but not used, and *external fragmentation*, when a sufficient number of free processors are available to satisfy a request, but they cannot be allocated contiguously.

Researchers have focused on *noncontiguous allocation* [2], i.e., the request of an application to be served by more than one contiguous partition, to reduce both types of fragmentation. In the past, noncontiguous allocations did not receive much attention because communication latencies were extremely sensitive to the physical distance in the network. However, depending on network characteristics and the program communication behavior, using distant free processors to serve a request may be more reasonable than denying the request [2].

1.2. Dynamic degree of parallelism

Dynamic behavior of parallel programs is another important issue to address. Currently, message passing interface-1 (MPI [3]), which does not support dynamic task creation, is still the dominant programming environment for parallel program development. Therefore, most of the presented allocation schemes have assumed that processor demand of a program is constant throughout execution time. This is an idealized and over-simplified assumption. Many parallel programming models and their corresponding language constructs allow dynamic creation and deletion of tasks, resulting in growing and shrinking demands. A wide range of parallel algorithms exhibit dynamic behavior in data access, workload and communication patterns. Examples can be found in radiosity calculations [4] and volumetric ray tracing [5] where workload and program flow depends on the way the light travels through modeled objects. Sparse Cholesky factorization as an alternative to the Gaussian elimination method [6,7] provides high performance by implementing dynamic behavior.

Although parallel virtual machine (PVM [8]) and SCI-based distributed shared memory systems (Scalable Coherent Interface [9]) already provide these functionalities, interest in MPI-2 based of systems supporting dynamic program behavior will increase. Consequently, processor allocation schemes will have to adapt to this development by providing dynamic partitions.

A partitioning scheme where partitions can 'breathe' will result in a better resource utilization. *Dynamic partitions* will minimize internal fragmentation, since the size of the partition will closely match the actual number of needed processors.

To completely avoid internal fragmentation, *form-free partitions* of arbitrary shapes could be used. However, even with form-free partitions, there will still be a considerable amount of external fragmentation, since 'holes' among the partitions will exist. In general, holes are not entirely detrimental, representing free space to allow partitions to breathe. If a partition wants to grow and there is no adjacent free space available, the request for more processors would be denied. A solution to this problem is to consider non-contiguous dynamic partitions.

If a large number of applications sharing a parallel machine and a high degree of dynamics is considered, i.e., frequent creations and deletions of tasks, a centralized allocation scheme could become a bottleneck due to the high number of partition size changes. This holds for the performance of the parallel application, too. Delays coming with the dynamic allocation may increase drastically at centralized allocation techniques in heavy load situations. In these scenarios, a distributed allocation scheme could be an alternative. Instead of sending request messages to a remote allocation agent, a distributed algorithm would search for free processors in the neighborhood to minimize overhead for allocation and keep the partition diameter low. Moreover, several searches for free processors by different programs could simultaneously take place, thereby avoiding the bottleneck problem presented by a centralized solution. The goal of this paper is to present the ideas supporting distributed strategies for the dynamic processor allocation problem and illustrate its usefulness.

1.3. Previous research work

Several approaches to dealing with the processor allocation problem can be found in the literature [10–13,2,14–18]. In spite of the fact that these contributions applied different policies in resource management, all schemes share one common characteristic: control of allocated resources is implemented by a global data structure localized mostly in a host machine. Implementation is relatively easy, and may be the most common approach. There are, however, problems associated with such a centralized management which may become relevant for large systems: (i) lack of scalability, (ii) incompatibility with adaptive processor allocation schemes (dynamic allocation) [19], and (iii) weak fault tolerance.

The scalability problem is caused by utilization of centralized structures in management. By increasing the number of processors to be managed, the global data structure grows, increasing processing time and reducing performance to a level that may not be acceptable for a procedure during execution time. In a centralized model, a *dynamic behavior* as described above would result in frequent updates to global data leading to an overhead in communication between the host and parallel machine. The host eventually becomes a bottleneck of I/O and computation. With regard to the fault tolerance problem, since all allocation operations have to go through the host, a host failure may stop all processing in the system.

Many of the previous policies also cause a considerable amount of machine fragmentation, a direct consequence of simplifications made by allocation schemes concerning partition shapes (rectangles) and contiguity restrictions. These simplifications reduce processing time of an allocation operation but increase both types of fragmentation, compromising overall machine utilization. Several alternatives must be considered when identifying a processor allocation scheme:

- static vs. dynamic;
- rectangular vs. form-free;
- contiguous vs. noncontiguous; and
- centralized vs. distributed.

In [20–22], we have already presented a distributed model for processor allocation with initial results for a structure preserving and a form-free distributed allocation scheme. We also analyzed the impact of noncontiguous allocation in a distributed scheme. In [23], we analyzed the feasibility of the dynamic allocation model in large PC clusters. We proposed and studied *Leak*, an enhanced noncontiguous version of one of our algorithms, as an alternative model for parallel programming that allows dynamic creation and task deletion. This paper summarizes our work with distributed allocation in multicomputers and presents our latest experimental results on three different cluster architectures.

The remainder of the paper is organized as follows: Section 2 introduces distributed allocation and presents a distributed allocation algorithm which, using simulation, is compared to other schemes in Section 3 with regard to contention, fragmentation and overhead. Measurements from a real cluster system are presented. Section 4 summarizes the obtained results.

2. Distributed processor allocation

Fig. 1 presents a global view of the distributed allocation model and the distributed *processor managers* (PMs) involved in the allocation operation. Main differences to centralized management are (i) the absence of a central data structure with information about the state of all processors, and (ii) the execution of allocation operations directly in the machine nodes in a distributed way, and not in a data structure localized in the host. The host machine is now only responsible for queuing incoming requests and forwarding them to the

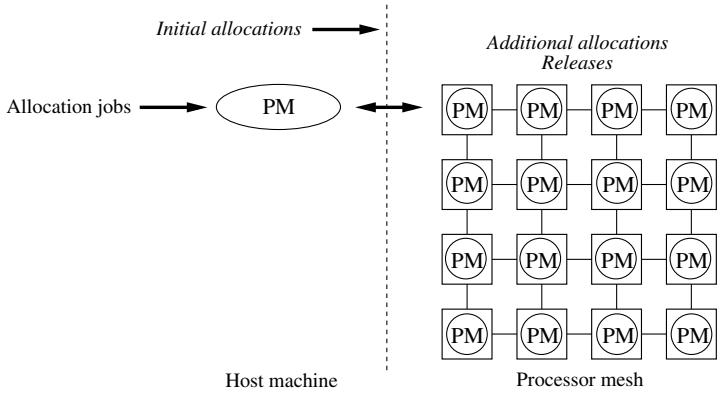

Fig. 1. Distributed allocation in a mesh-connected multicomputer.

processor mesh. Communication between host and machine exists through a boundary node. This node is called an *entry point* and due to the distributed environment several entry points may be used to improve scalability.

Each machine node has a local PM that is responsible for the processor allocation. The PMs cooperate to solve the allocation problem in a distributed way [24]. Communication among the PMs and the implemented distributed allocation scheme depends heavily upon the architecture of the target machine. To simplify the search for free resources, PMs form a logical topology to efficiently map the interconnection network of the target machine. Section 2.3 describes in detail how PMs cooperate in our target system.

2.1. Processor allocation operations

To match the distributed characteristics of this new allocation model the basic allocation operations are adapted and new dynamic allocation operations are implemented in the distributed processor manager, resulting in allocation operations being divided into two groups:

1. *Static operations*
   Initial allocation Allocate the processors needed by a parallel application to initiate execution.
   Final release Release all processors after execution.
2. *Dynamic operations*
   Partial allocation Allocate extra processors needed during execution to expand a processor partition already in use.
   Partial release Release some processors to shrink a processor partition already in use.

Initial allocation is the most costly operation in the distributed environment. It originates in the host computer and initiates a search wave in the machine for the desired partition. Since all the nodes are possible candidates, the search scope may be large. The first-fit strategy is used in the search and different initial nodes are used each time as mesh entry-points to increase the probability of finding free nodes in early stages of the search wave.

In contrast to centralized list-based algorithms (released processors may have to be concatenated to a free partition or will concatenate multiple free partitions in one), the release operation in distributed allocation is trivial. Starting in one of the partition nodes, a wave is used to change the state of the involved nodes from *allocated* to *free*. No data structure has to be updated and the operation is done completely inside the machine, initiating from within the partition to be released.

Dynamic operations allow a running parallel application to allocate additional processors and dynamically adapt the partition size in use to a new processor demand ('breathe'). To begin, the application sends a partial allocation request to the PM of one of its nodes. In partial allocation a search wave for free processors will originate in this node looking for possible candidates to be allocated around this partition. In a partial release, the wave searches for the nodes to be liberated. As a result of the smaller search scope, both operations generate fewer messages than static operations.

To eliminate problems like not finding free adjacent nodes in a partial allocation (a partition has no surrounding space in which to grow) or having released nodes that are inside a partition and will not be of much use because they will not be adjacent to any other partition, we only use dynamic operations in conjunction with noncontiguous allocation.

2.2. Noncontiguous allocation

In most of the distributed processor allocation algorithms presented in [22], external fragmentation was high (up to 30%), leading to compromised machine utilization. The main reason for this behavior is that these algorithms use contiguous allocation to reduce resulting partition diameters, with intent to reduce communication costs inside the partition. Current communication technologies like wormhole routing [25] enable us to consider noncontiguous allocation schemes, since the number of hops between nodes is not the dominant factor determining message latency [2]. Use of small partitions of free processors scattered throughout the machine to form larger noncontiguous partitions significantly decreases external fragmentation. However,

in ring connected machines like SCI clusters [9], or in machines with hierarchical networks, noncontiguous allocation introduces potential problems of message contention because messages occupy more links, yielding potential communication interference with other jobs. If contiguity is an issue, we try to serve a request with contiguous allocation, and to look for noncontiguous additions only on demand. This way the noncontiguous scheme should be seen as an addition, and not as an alternative to contiguous allocation. In some of our target machines contiguity is not an issue anymore because they use some kind of switched network where the distance between any two nodes is 1.

2.3. Distributed Leak algorithm

Ideas presented in this paper are validated by implementation of a distributed allocation algorithm called Leak (Section 3.2). It was developed for contiguous allocation in 2D torus interconnected systems. Section 3.2 also provides a detailed description of this machine.

The Leak algorithm is based on the principle of leaking water. From an origin point, an amount of water leaks and flows to directions where no resistance is encountered. An important element to keep in mind is that leaking water exhibits cohesion, which limits the diameter of the resulting puddle. For a distributed processor allocation, the number of processors to be allocated corresponds to the amount of leaking water. Processors already allocated in the mesh are the resistance areas and the final area formed by the allocated processors is the resulting puddle.

```
WHILE (true) DO
   wait for search_initial_message(load, requester)
   IF (node is free)
      initial = myself
      send it allocate_message(initial, load) to myself // starts allocation
      wait for confirmation_message(remaining_load)
      IF (remaining_load >0 )
         IF (not last node in sequence)
            send search_initial_message(remaining_load, requester) to next in sequence
            // increase/decrease column or/and increase/decrease line depending on origin point
         ELSE
            send(`allocation failed`) to requester
            send release_message() to node_list
         ENDIF
      ELSE
         send(`allocation succeeded`) to requester
      ENDIF
   ELSE
      IF (not last node in sequence)
         send search_initial_message(remaining_load, requester) to next in sequence
      ELSE
         send(`allocation failed`) to requester
         send release_message() to node_list
      ENDIF
   ENDIF
ENDWHILE
```

Fig. 2. Phase 1 of an initial allocation: search for initial node.

```
WHILE (true) DO
   wait for allocate_message(initial, load)
   set node to occupied and add to node_list
   decrement load
   IF (load == 0)
      send confirmation_message(load) to initial
   ELSE
      no_of_free_neighbors = check for free neighbors()
      avg_load = load / no_of_free_neighbors
      send allocate_message(myself, avg_load) to all free neighbors
      wait for confirmations // 0 if succeeded, any other number is remaining load
      accumulate remaining_load and combine node_lists
      send confirmation_message(remaining_load, node_list) to initial
   ENDIF
ENDWHILE
```

Fig. 3. Phase 2 of an initial allocation: recursive distributed allocation.

In Leak, an initial allocation is composed of two phases. In the first phase (Fig. 2) machine nodes are searched from a origin point using a sequential search wave (in the used mesh topology the nodes are searched with a 'snake' pattern until all rows are traversed and all four corners are eligible origin points). Since this operation originates outside of the machine, we refer to the origin points as *entry points*. Depending on the location of the entry point the orientation of the search pattern is adapted (left-right, top-bottom). In the second phase (Fig. 3) all direct neighbors of the origin point are tested in parallel to determine if any are free. Each free neighbor then becomes part of the load and the second phase continues recursively and in parallel until no more load is available. All nodes found free are tried as origin points until a free partition of suitable size is found or no more nodes are available to try, and the allocation is denied.

Fig. 4 exemplifies the execution of a 4-processor request. Gray areas represent already allocated partitions and striped nodes are involved in the ongoing allocation operation. After a feasible origin point is found with a search wave (Fig. 4a), possible flowing directions are determined and remaining load is redistributed (Fig. 4b, c). Numbers and arrows indicate how the load is being propagated from the request across the allocated nodes. This procedure is repeated recursively until all processors are allocated.

The essential feature of the algorithm is its form-free allocation strategy, i.e., partitions are no longer restricted to rectangles, but may have assume any arbitrary shape, allowing the processor management more flexibility to find a partition of suitable size, and resulting in less fragmentation. Due to the recursive nature of the algorithm and its distributed execution, it is also important to note that different flowing directions allocate processors in parallel, resulting in a reduced allocation time. The parallel potential of an allocation operation increases with the size of the requested partition. Fig. 5 exemplifies this behavior in the execution of a 5-processor request by a noncontiguous version of the *Leak* algorithm. Under a contiguous scheme, the entry point from Fig. 5a and its rightmost neighbor would be released after verifying that no more flow directions were available, and the search wave would continue for a suitable origin point. But in a noncontiguous scheme the initially requested load would be decreased by each partially successful allocation (Fig. 5b, c).

Partial allocation (in dynamic versions of the algorithm) is similar to an initial allocation (with two phases). Two main differences exist: (i) the origin point in phase 1 is not a machine entry point but a node inside the allocated partition, and (ii) the sequential search wave will have a spiral pattern since free nodes near the origin point are preferred. Which node of the partition will be used as an origin point depends upon the application since each node of an allocated partition may request a partial allocation (in a master slave application it is usually done by the node where the master process resides).

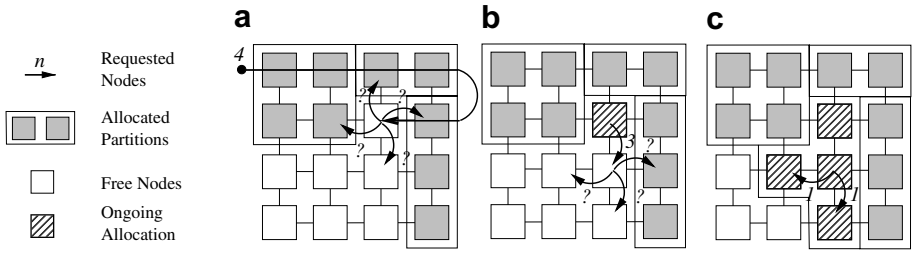

Fig. 4. Contiguous Leak algorithm.

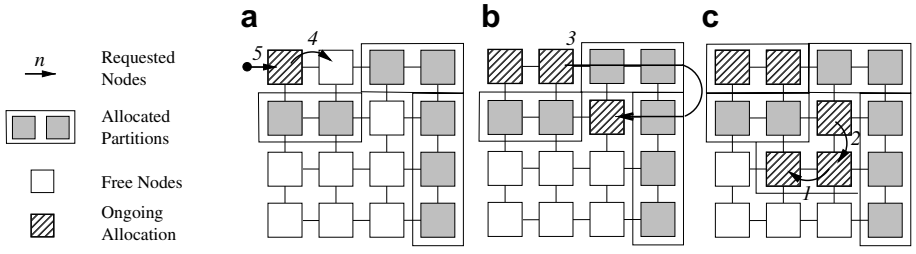

Fig. 5. Noncontiguous Leak algorithm.

3. Performance analysis

The following experiments were conducted to investigate the potentiality and feasibility of the proposed dynamic allocation in multicomputers:

1. Fragmentation experiments;
2. Message-passing contention experiments;
3. Intermittent service requests experiments; and
4. Allocation overhead experiments.

Our discrete event simulator [20] is a multicomputer simulator supporting experimentation with distributed allocation strategies on architectures with mesh-connected network topologies. The simulator evaluates the effects of system fragmentation and generated allocation messages. It was used to study fragmentation generated by dynamic distributed allocation in machines with up to 1024 nodes (item 1 of the above list). Experiments 2, 3 and 4, were executed in a real system, a mesh-connected *Scalable Coherent Interface* cluster (SCI [9]).

3.1. Simulated fragmentation experiments

This set of experiments, studying the effects of fragmentation on system utilization and job response time, are modeled after the simulation experiments conducted in previous allocation strategy research [2,11]. In these experiments, jobs arrive, are scheduled with FCFS, delay for an amount of time taken from an exponential distribution, and then depart. Allocation messages are also modeled, to evaluate the message overhead in the distributed allocation.

In these experiments we simulated a dynamic and a static version of the distributed noncontiguous form-free Leak algorithm [23,22], and the contiguous structure preserving frame sliding (FS) [10]. The distributed Leak algorithm was described in Section 2.3. FS examines the first candidate 'frame' from the lowest leftmost available processor and slides the candidate frame horizontally or vertically by the stride of width or height of the requested submesh, respectively, until an available frame is found or all candidate frames are checked. Because FS only handles rectangular requests, the job size in this case was transformed to the best possible rectangle with regard to internal fragmentation and partition diameter.

The independent variable in these experiments was the system load, defined as the ratio of the mean service time to mean interarrival time of jobs. Higher system loads reflect greater demands when jobs arrive faster than they can be processed. Jobs delay for an exponentially distributed service time with a mean of 1.0 time units. For example, under a system load of 1.0, jobs arrive as fast as they are serviced, on average, and under a system load of 2.0, jobs arrive twice as fast as they can be serviced.

Job request size is randomly generated from one of two different distributions, uniform and exponential. In the uniform distribution, the size of each job is uniformly distributed over the range $U[a,b]$, with $a = 1$ and b having four times the side length of the entire mesh. In the exponential distribution, job size is exponentially distributed with a mean of twice the side length of the entire mesh. In this case, there are many small jobs and fewer large ones. To simulate the dynamic behavior of parallel applications, four load profiles are randomly generated for each job: constant (a), increasing (b), decreasing (c) and pyramid (d). In the constant profile, processor demand is static, not varying during execution. In the increasing and decreasing profiles the processor demand varies from 1 to job size and from job size to 1, respectively, during execution. The pyramid profile simulates divide-and-conquer algorithms, with the processor demand increasing from 1 to job size in the first half of the execution time and decreasing to 1 in the second half. In our simulations, 30% of the jobs have a dynamic load profile (each of the three variants are equally represented in this percentage).

Each job size distribution experiment measures:

Finish time (Ft): the time required for completion of all the jobs.
Job response time (Jrt): the time from when a job arrives in the waiting queue until the time it is completed.
System utilization (Su): the percentage of processors that are utilized over time.

Table 1
Fragmentation experiments for a heavy system load (10.0)

Algorithms	Distribution	Ft	Jrt	Su (%)	Mpa
Dynamic leak	Uniform	185	57.73	96.85	750.3
	Exponential	157	32.75	97	507
Static leak	Uniform	266	99.07	60.49	5949
	Exponential	180	43.82	63.72	2184
Frame sliding	Uniform	357	152.85	47.82	1083
	Exponential	243	72.53	50.45	849

Messages per allocation (Mpa): the total number of generated messages to allocate incoming requests from the processor management divided by the number of generated requests.

All simulations model a 32×32 mesh and run until 1000 jobs have been completed. Results reported for fragmentation experiments represent the statistical mean after 10 simulation runs with identical parameters, a 95% confidence level, and mean results with less than 5% error.

Table 1 shows how efficiently the three algorithms handle a system saturated by job requests with job sizes taken from each distribution. Simulation results for a heavy system load of 10.0 are presented. At this load, the system waiting queue is filled very early in the simulation (full load), allowing each allocation strategy to reach its upper limits of performance.

As expected, we can see the dynamic strategy achieved the highest system utilization since it is the only strategy that can cope with dynamic processor requests. Static strategies must allocate fixed partitions with the highest number of needed processors, increasing internal fragmentation. The dynamic strategy also profits from allocating form-free noncontiguous partitions. Bigger partitions are difficult to find in contiguous schemes resulting in long search waves and several attempts, each increasing the number of messages and response time. In a noncontiguous scheme, allocations are cumulative resulting in shorter search waves and reduction of messages and time. The difficulties of structure-preserving contiguous allocation can be verified with the FS strategy and its resulting poor system utilization.

Fig. 6 shows the average job response times for uniform job size distributions at varying system loads. Since the response time reflects the ability of each strategy to find free nodes in the machine to satisfy an incoming request, the form-free Leak algorithms achieve, as expected, the lowest times. For the same reason, the advantage of the noncontiguous Leak version is also significant, especially at high system loads. Note that the response times do not increase to infinity when the system load appraoches saturation as one would expect. This is due to the fact that we have only a finite job stream, i.e., the response times are bound by the overall finish time of the simulation.

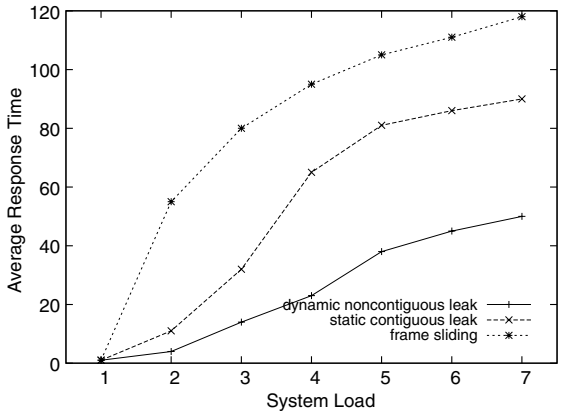

Fig. 6. Average job response time vs. system load for uniform distribution of job sizes.

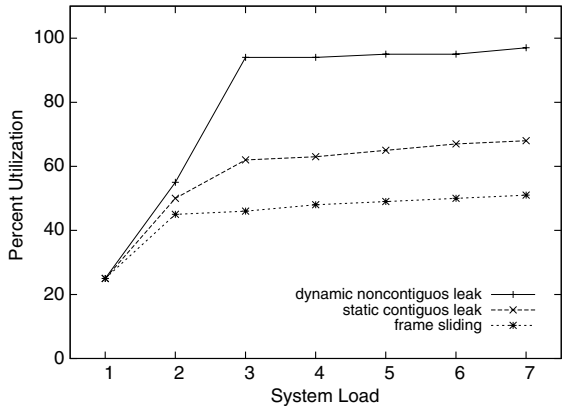

Fig. 7. System utilization vs. system load for uniform distribution of job sizes.

Fig. 7 plots system utilization and job size distribution at varying system loads for the same algorithms. All three strategies reached peak utilization at system loads of approximately 3.0. The noncontiguous Leak reached up to 97% utilization, whereas the contiguous Leak reached only 65%, due to fixed partitions. The structure preserving FS was only able to reach 51% because of high internal and external fragmentation. Results reflect how each of the algorithms cope with both types of fragmentation. FS suffers from internal and external fragmentation and the form-free Leak algorithms eliminate internal fragmentation for jobs with static load profiles. The results clearly illustrate that the only strategy that can efficiently handle jobs with dynamic load profiles is the dynamic Leak variant. It eliminates all types of internal fragmentation and, because it is noncontiguous, also significantly reduces external fragmentation.

It is important to note that a dynamic algorithm may not always obtain the highest throughput and lowest job response time compared to static strategies. With high load, the optimistic approach of the dynamic allocation (no reservations are made for possible future increases in the number of processors) may result in partitions that are allocated but do not have space in which to grow. The processing time of these partitions must be extended, increasing response time and reducing machine throughput.

Results measured in these experiments are consistent with those reported by Zhu [11] for the contiguous FS strategy (mean system utilization around 50%) and by Lo [2] for the FS and noncontiguous strategies (mean system utilization by a noncontiguous scheme over 75%). The distributed Leak strategy reached a system utilization over 90% because it uses form-free allocation in combination with noncontiguity to reduce internal fragmentation.

These fragmentation experiments indicate that, for variable workloads, dynamic allocation is far superior to static in terms of its ability to utilize processors. However, these results ignore increased communication contention that may be introduced as a result of noncontiguous allocation (our dynamic algorithm relay on noncontiguous allocation especially for additional requests). Therefore, in order to validate dynamic allocation as a viable strategy, experiments in a real system must be performed to evaluate message contention.

3.2. Distributed allocation on a real system

The following three experiments were conducted on the Siemens hpcLine Primergy High Scalable Compute Server at the Paderborn Center for Parallel Computing (PC^2) with a prototype of the PM described in Section 2 as a first step to evaluate the feasibility of the dynamic allocation on real parallel machines:

Message-passing contention experiments Network contention plays an important role in the proposed allocation since allocation messages could be stealing bandwidth from ongoing applications. If a so-called secondary network is not available – usually a Fast-Ethernet network for program loading, management and monitoring purposes – allocation messages generated by processor managers will share the same network with the parallel applications that are executing in the machine. Depending on the available network bandwidth,

usually one of the main bottlenecks in cluster computing, this interference may lead to delays in an applications' execution time. In this experiment, contention effects are analyzed based on worst case scenarios.

Intermittent service requests experiments In the proposed distributed allocation, nodes which participate in ongoing computations in the machine will also have intermittent service requests to their local processor manager. Each request generates allocation messages that are sent to machine nodes to look for free processors. These messages will awaken the local PM in each node so that it will compete for processor time with other local tasks. This experiment investigates the impact this might have on overall machine performance. Evidence in the literature suggests that asynchronous interruptions in nodes can seriously deteriorate parallel machine performance [26].

Allocation overhead Because of the distributed nature of the proposed allocation, latency requests are higher than in centralized strategies. This experiment measures the latency of distributed allocation operations for each target machine under different system workloads.

The hpcLine Primergy Server is a distributed memory multicomputer with 96 nodes (two Intel Pentium II with 450 MHz and 512 MB DRAM) connected by a two-dimensional SCI torus [9]. Each machine is connected to the network through two SCI rings, one horizontal and one vertical. The routing is configured as XY with messages traversing first in the horizontal ring (X) of the origin node until they reach the vertical ring (Y) of the destination node. The SCI rings are unidirectional and each one has a total bandwidth of 500 MB/s. This bandwidth is shared by all nodes in the same ring. The SCI cards in each node are PCI-SCI (32 bits, 33 MHz PCI bus) adapters, model D312 (distributed with Scali Wulfkits), equipped with SCI link controller LC2 and PSB revision D. Programs were written in a special MPI version for the SCI hardware (ScaMPI [27]) and run on the Solaris operating system (Fig. 8a).

Fig. 8b presents the logical allocation structure chosen for this machine. Because SCI channels provide high bandwidth, allocation operations are executed in the SCI network. Allocation operations attempt to allocate form-free contiguous partitions to reduce message interference in neighbor nodes, allowing noncontiguous allocation if necessary. The PM implements the Leak algorithm described in Section 2.3 and was coded with the C language. Low-level SCI routines are used for communication purposes. One PM has approximately 1500 lines of source code and the executable has only 45 kB. The host machine is responsible for forwarding requests to the cluster and evaluating results. All four corners of the machine alternate as entry points to initial allocation search waves.

In order to better quantify the effects of contention, a simple worst-case contention generating program that is able to send multiple messages over the same physical ring was developed. Our program allocates nodes in the topmost mesh row (12 nodes) and in the rightmost column (7 nodes, excluding the common node). All

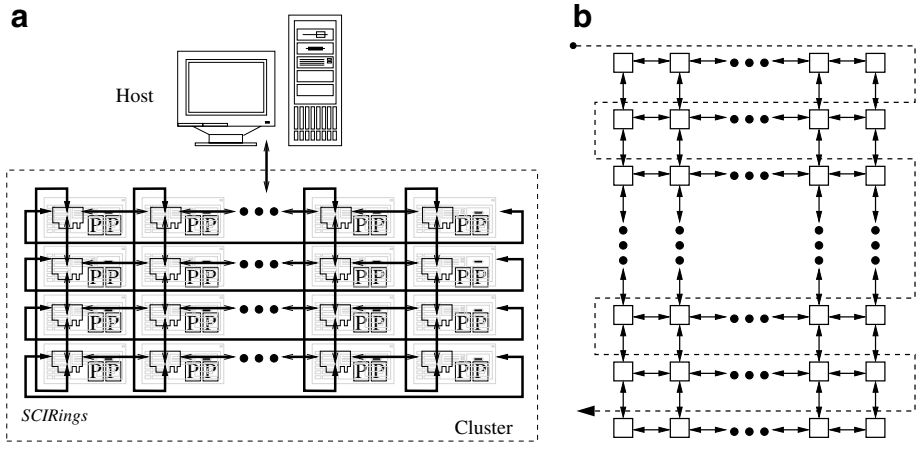

Fig. 8. Primergy high scalable computer server (SCI network): machine architecture (a) and logical allocation structure (b).

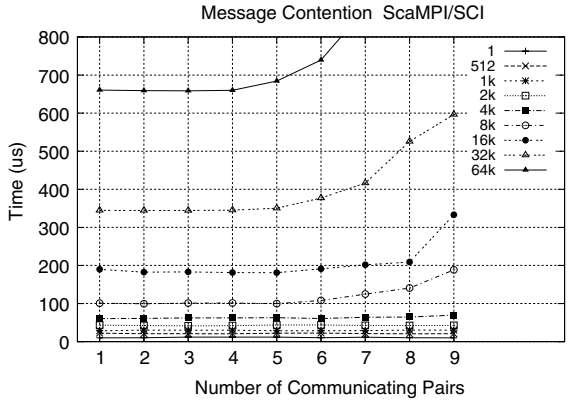

Fig. 9. Worst case contention in the Siemens Primergy Server.

Table 2
Allocation time in the Primergy multicomputer

Algorithm	Mean allocation time (s)	Generated messages per allocation
Dynamic noncontiguous leak	0.03	23
Static noncontiguous leak	0.162	128

nodes are paired from the horizontal node 10 (not involved) outward and exchange messages with each other allowing up to nine pairs of communicating nodes while continuing to share the same horizontal ring from the machines XY routing. Fig. 9 presents obtained communication latency between horizontal nodes 9 and 11 for message sizes up to 64 kB and an increasing number of involved communicating pairs.

Virtually no contention is noticeable for messages smaller than 4 kB. For larger messages, contention begins to slow down message-passing performance, but only for more than 6 pairs of communicating nodes. These results are better than the ones measured in [2] for the Intel Paragon with a similar contention program due to the better bandwidth of the Primergy mesh. This empirical data are encouraging, indicating that small messages generated by the distributed dynamic allocation (100 bytes mean size) will not affect performance of ongoing computations. It also indicates that noncontiguous allocation that may separate communicating pairs in a noncontiguous partition, is feasible on this machine.

In regard to intermittent requests on machine nodes, we were unable to measure any significant performance degradation (less than 5%) on real applications running simultaneously with the distributed management compared with the same applications running in a dedicated system. Small allocation messages (mean size is 100 bytes) and low complexity of local processor managers are the main reasons for these results.

To measure allocation overhead, we simulated incoming requests (the same load generation module of the simulator from Section 3.1 was used). Only 64 nodes connected as an 8×8 torus were used for this experiment. Incoming parallel jobs were not actually loaded on the machine and allocated partitions were only reserved during job duration. In our performance test for a medium system load (5.0), we obtained allocation times of around 0.03 s for dynamic allocation and 0.162 s for static allocation (Table 2). Due to the small search scope of the partial allocation operation in the dynamic version of the algorithm we observed that the number of generated messages per allocation is much smaller than in the static version, leading to a reduction in the average allocation time.

4. Conclusions

This paper proposes a dynamic distributed processor allocation for multicomputers with support for growing and shrinking processor demands. In our distributed allocation model, the central entity responsible for processor status control is eliminated and allocation operations are executed in parallel in the processor network itself. Using this approach, no centralized status of the processors in the entire machine exists, and each

node participates in the allocation of partitions through its PM, communicating and cooperating with other nodes. Nodes may be allocated or released during the execution of parallel programs and requests may originate in any machine nodes.

The basic allocation operations were redefined to match characteristics of this new dynamic environment and implemented in a distributed processor manager. We presented simulation results and measurements using a PMs prototype implemented for a 96 nodes mesh-connected SCI (Scalable Coherent Interface) cluster.

Our study shows that the dynamic distributed approach is feasible for large cluster machines with current communication technologies, permits greater parallelization of allocation operations, eliminates bottlenecks of the centralized approach, and achieves a much higher processor utilization. In our simulations, system utilization for the noncontiguous version of one of our dynamic algorithms reached as high as 97%.

We conclude that distributed dynamic allocation provides a new approach that will assist highly parallel systems to achieve better price/performance ratios in high demand, multi-user environments. New models that support this paradigm, like the MPI-2 process model that allows the creation and cooperative termination of processes after an MPI application has begun, will be able to exploit the potential of the distributed dynamic allocation.

The general structure of the proposed algorithms with its distributed scheme of allocating resources can also serve as a blueprint for other allocation or reservation problems in distributed systems, i.e., bandwidth reservation in large networks. We feel that with new services based on the Internet, new types of allocation/reservation problems for different resources will arise that need dynamic distributed algorithms similar to the algorithms proposed in this paper.

References

[1] D.G. Feitelson, A survey of scheduling in multiprogrammed parallel systems, Technical Report RC 19790, IBM Research Division Research Report, 1994.
[2] V. Lo et al., Noncontiguous processor allocation algorithms for mesh-connected multicomputers, IEEE Transactions on Parallel and Distributed Systems 8 (7) (1997) 712–726.
[3] Message Passing Interface Forum MPIF. MPI: A Message–Passing Interface Standard, Technical Report, University of Tennessee, Knoxville, 1995.
[4] J.P. Singh, C. Holt, T. Totsuka, A. Gupta, J. Hennessy, Load balancing and data locality in adaptive hierarchical n-body methods: Barnes-hut, fast multihole, and radiosity, Journal of Parallel and Distributed Computing (27) (1995) 118–141.
[5] S. Frank, A. Kaufman, Dependency graph scheduling in a volumetric ray tracing architecture, Graphics Hardware (2002) 1–9.
[6] E. Rothberg, A. Gupta, An evaluation of left-looking, right-looking, and mutlifrontal approaches to sparse cholesky factorization on hierarchical – memory machines, International Journal High Speed Computing (5) (1993) 537–593.
[7] G. Runger, T. Rauber, C. Scholtes, Scalability of sparse cholesky factorization, International Journal High Speed Computing (10) (1999) 19–52.
[8] Al. Geist et al., PVM: Parallel Virtual Machine, MIT Press, Cambridge, 1994.
[9] IEEE. IEEE standard for scalable coherent interface (SCI). IEEE 1596–1992, 1992.
[10] B. Melhart, C. Morgenstern, T. Nute, A compendium of processor allocation strategies for two-dimensional mesh connected systems, Concurrency: Practice and Experience 5 (7) (1995) 497–514.
[11] Y. Zhu, Fast processor allocation and dynamic scheduling for mesh multicomputers, International Journal of Computer Systems Science and Engineering 2 (11) (1996) 99–107.
[12] D.D. Sharma, D.K. Pradhan, Processor allocation in hypercube multicomputers: fast and efficient strategies for cubic and noncubic allocation, IEEE Transactions on Parallel and Distributed Systems 10 (6) (1995).
[13] Geunmo Kim, Hyunsoo Yoon, On submesh allocation for mesh-connected multicomputers: a best-fit allocation and a virtual submesh allocation for faulty meshes, IEEE Transactions on Parallel and Distributed Systems 9 (2) (1998).
[14] Ge-Ming Chiu, Shin kung Chen, An efficient submesh allocation scheme for two-dimensional meshes with little overhead, IEEE Transactions on Parallel and Distributed Systems (1999) 471–486.
[15] Po-Gen Chuang, Chih-Ming Wu, An efficient recognition-complete processor allocation strategy for k-ary *n*-cube multiprocessors, IEEE Transactions on Parallel and Distributed Systems (2000) 485–490.
[16] Hyunseung Choo, Seong-Moo Yoo, Hee Yong Youn, Processor scheduling and allocation for 3D torus multicomputer systems, IEEE Transactions on Parallel and Distributed Systems (2000) 475–484.
[17] M. Livingston, Q. Stout, Fault tolerance of the cyclic buddy subcube location schemes in hypercubes, 1991.
[18] F. Wu, C. Hsu, L. Chou, Processor allocation in mesh multiprocessors using the leapfrog method, IEEE Transactions on Parallel and Distributed Systems 14 (3) (2003) 276–289.
[19] V.K. Naik, S.K. Setia, M.S. Squilante, Performance analysis of job scheduling policies in parallel supercomputing environments, In Supercomputing 1993, 1993, pp. 824–833.

[20] C.A.F. De Rose, Distributed processor management in multicomputers, Phd Thesis (in German), University of Karlsruhe, Germany, 1998.
[21] C.A.F. De Rose, H-U. Heiss, P. Navaux, Distributed processor allocation for large PC clusters, In: Ninth International Symposium on High Performance Distributed Computing, 2000, pp. 288–289.
[22] C.A.F. De Rose, P.O.A. Navaux, C. Geyer, Distributed processor allocation in mesh-connected multicomputers, In: PDPTA 2000, 2000, pp. 1191–1197.
[23] C.A.F. De Rose, H-U. Heiss, Dynamic processor allocation in large mesh-connected multicomputers, In: EURO–PAR 2001, 2001, pp. 783–792.
[24] N.A. Lynch, Distributed algorithms, The Morgan Kaufmann Series in Data Management System, California, USA, 1996.
[25] L.M. Ni, P.K. McKinley, A survey of wormhole routing techniques in direct networks, IEEE Transactions on Computers, 1993.
[26] T. Tabe, J. Hardwick, Q. Stout, Statistical analysis of communication time on the IBM SP2, Computing Science and Statistics (27) (1995) 347–351.
[27] Scali. Scampi release notes, <http://www.scali.com>.