Dynamic Processor Allocation in Large Mesh-Connected Multicomputers*

César A. F. De Rose[1] and Hans-Ulrich Heiss[2]

[1] Catholic University of Rio Grande do Sul, Computer Science Department
90619-900 Porto Alegre, Brazil
derose@inf.pucrs.br
[2] University of Paderborn, Department of Computer Science
33098 Paderborn, Germany
heiss@upd.de

Abstract. Current processor allocation techniques for highly parallel systems are based on centralized front-end based algorithms. As a result, the applied strategies are restricted to static allocation, low parallelism and weak fault tolerance. To lift these restrictions we are investigating a distributed approach to the processor allocation problem in large mesh-connected multicomputers. A noncontiguous version of a distributed dynamic processor allocation strategy is proposed and studied in this paper as an alternative for parallel programming models that allow dynamic creation and deletion of tasks. Simulations compare the performance of the proposed dynamic strategy with the static counterpart and also with well-known centralized algorithms in such an environment with growing and shrinking processor demands. We also present the results of experiments on a Siemens hpcLine Primergy Server with 96 nodes that show dynamic allocation is feasible with current technologies.

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 some kind of network to exchange messages. Despite some specific applications where a program is running permanently on a dedicated machine, it is almost inevitable in large systems with hundreds or thousands of nodes, to allow *multiprogramming*, i.e. many parallel programs share the machine in space in order to achieve high machine utilization. We assume that upon arrival, each program requests a specific number of processing nodes. Such a request is usually satisfied by allocating a sufficiently large *partition* of the processors to the program. *Processor allocation* involves the selection of a partition for a given parallel job, with the goal of maximizing throughput over a stream of many jobs. A resource management scheme for processor allocation has to meet several partly contradicting goals:

* This research was supported in part by HP-Brazil and Fapergs.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 783–792, 2001.

High utilization It should maximize the utilization of the resources, i.e. it has to avoid any kind of fragmentation so that all processors can be used.

Appropriate shapes It should support low execution times of the parallel programs. The execution time will be affected by the allocation scheme with regard of the communication bandwidth and latencies within the partition (in a 2D-mesh, a partition that forms a square would better serve an arbitrary program than a partition shaped as a narrow and long stripe).

Low overhead Since all requests are processed at run-time, the resource allocation algorithms have to be fast and should cause only low overhead.

Scalability The algorithms should be able to support systems of thousands of nodes without becoming a bottleneck.

In the following, we constrict our work to multicomputers connected by a two-dimensional mesh since most of the currently existing MPPs and also cluster computers connected with SCI boards [3] are based on 2D- or 3D-meshes (the extension of our 2D-algorithm to the 3D-case is rather straightforward).

2 Processor Allocation Policies

Because allocation operations need to be fast, usual allocation techniques restrict the feasible shapes of partitions to achieve some regularity, which facilitates their 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. In our case of 2D-meshes it means that always rectangular submeshes are allocated. In addition, most systems also require that the allocated processors are constrained to be physically adjacent (*contiguous* allocation). So each request will be served by exactly one rectangular partition of sufficient size. When using rectangles, however, a 100% utilization of the processor resource is impossible due to two types of *fragmentation*: internal fragmentation, when processors are allocated, but not used, and external fragmentation, when there are free processor partitions that cannot be allocated since they are too small.

Another important point is the dynamic behavior of parallel programs. Previously presented schemes have assumed that the processor demand of a program is constant through its execution time. This is an idealized or 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 partitioning scheme where partitions can "breathe" will result in better utilization. Dynamic partitions will minimize the internal fragmentation, since the size of the partition closely follows the number of processors actually needed. This is difficult to achieve when we stick to rectangular partitions because we only could add or remove some boundary rows or columns which again would result in some internal fragmentation. To completely avoid internal fragmentation, free-form partitions have to be used which can be of arbitrary shape. However, even with free-form partitions, there will be still a considerable amount of external fragmentation, since there will be "holes" between the partitions. Holes in general are not completely bad, since they represent free space

that allows the partitions to breathe. If a partition wants to grow and there is no adjacent free space available, the request for more processors has to be denied. A solution to this problem could be to resort to noncontiguous allocation, i.e. the request of an application will be served by more than one contiguous partition.

Several approaches to deal with the processor allocation problem can be found in the literature [9,4,5,1,2]. In spite of the fact that they apply different policies in the resource management, all the schemes have one in common: the control of allocated resources is done with a global data structure localized mostly in a host machine. This is easy to implement and may be the natural approach. There are, however some problems associated with such a centralized management which may become important for large systems: (i) lack of scalability, (ii) the incompatibility with adaptive processor allocation schemes [6] (dynamic allocation), and (iii) it's weak fault tolerance. The scalability problem is caused by the utilization of centralized structures in the management. By increasing the number of processors to be managed, the global data structure grows, increasing its processing time and reducing the performance to a level that may not be acceptable for a procedure done at execution time. In a centralized model, a *dynamic behavior* as described above would result in frequent updates to the global data leading to an overhead in communication between host and parallel machine. The host eventually becomes a bottleneck of both I/O and computation of the parallel machine. Regarding the fault tolerance problem, since all allocation operations have to go through the host, a host failure may stop all processing in the system.

Most of the previous policies cause also a high machine fragmentation. This is a direct consequence of the simplifications made by the allocation schemes concerning the shape of the partitions (rectangles) and the restriction to contiguity. These simplifications reduce the processing time of an allocation operation but increase both types of fragmentation, compromising the overall machine utilization. To summarize, there are several alternatives when considering a processor allocation scheme: static vs. dynamic, rectangular vs. free-form, contiguous vs. noncontiguous and centralized vs. distributed.

In [8] we have already presented a distributed model for processor allocation with some initial results for a structure preserving and a free-form distributed allocation scheme. We also analyzed the impact of noncontiguous allocation in a distributed scheme. In this paper, we analyze the feasibility of the dynamic allocation model in large PC clusters. We propose and study an enhanced noncontiguous version of one of our algorithms, called *Leak* as an alternative for parallel programming models that allow dynamic creation and deletion of tasks. We consider the use of a distributed implementation as rather natural for this type of environment, however, centralized implementations are also possible.

3 Distributed Processor Allocation

Figure 1 shows a global view of the distributed allocation model [8] and the distributed *Processor Managers* involved in the allocation operation. The main

differences to the 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 processor mesh in a distributed way, and not in a data structure localized in the host. The host machine is now only responsible for queuing the incoming requests and forwarding them to the processor mesh. Each node in the mesh has a local Processor Manager (PM) responsible for the processor allocation. The PM's cooperate to solve the allocation problem in a distributed way.

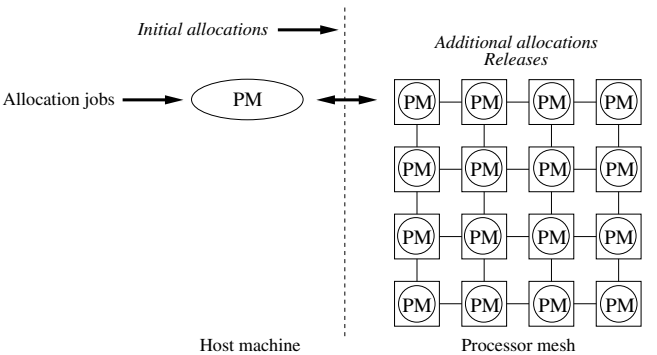

Fig. 1. Distributed allocation

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. This results in allocation operations being divided in two groups: static (initial allocation and final release) and dynamic (partial allocation and partial realease). The initial allocation is the most costly operation in the distributed environment. It originates in the host computer and initiates a search wave in the mesh for the desired partition. Since all the mesh nodes are possible candidates, and we are considering large machines with many nodes, the search scope is very 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 an free partition or will concatenate multiple free partitions in one), the release operation is trivial in a distributed management. Starting in one of the partition nodes, a wave is used to change the state of the involved processes to *free*.

The dynamic operations allow a running parallel application to allocate additional processors and to adapt the partition in use dynamically to a new processor demand (breath). To start this operation, the application sends a local allocation request to the PM of one of its nodes. A search wave for free processors will be originated in this node and will search for possible candidates around this

partition in the case of a partial allocation or, in the case of a partial release, for specific nodes to be liberated. Both operations generates much fewer messages than the static operations due to the smaller search scope.

Distributed Allocation Algorithm The implemented PM from section 3 uses an enhanced version of the Leak algorithm [8]. This algorithm is based on the principle of leaking water. From an origin point, an amount of water leaks and flows to the directions where no resistance is encountered. The algorithm has two phases. In the first phase a suitable origin point is searched with a sequential search wave (in the used mesh topology the nodes are searched from left to right in each row until all rows are traversed). In the second phase all the direct neighbors of the origin point are tested in parallel if they are free. Each free neighbor 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 point until a free partition of suitable size is found or no more nodes are available to try and the allocation is denied. Figure 2 exemplifies the execution of a 4-processor request. After a feasible origin point is found with a search wave (Figure 2a), the possible flowing directions are determined and the remaining load is distributed (Figure 2b-c). This procedure is repeated recursively until all processors are allocated.

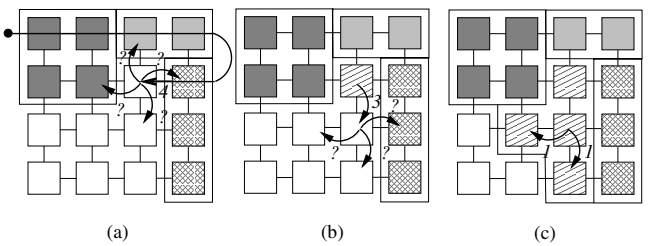

Fig. 2. Contiguous Leak algorithm

The essential feature of the algorithm is its free-form allocation strategy, i.e. partitions are no longer restricted to rectangles, but may have an arbitrary shape. This gives the processor management more flexibility to find a partition of suitable size, and results in less fragmentation. Due to the recursive nature of the algorithm and its distributed execution in the machine, it is also important to notice 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.

Current communication technologies like wormhole routing [7] enable us to consider noncontiguous allocation schemes, since the number of hops between nodes is not the dominant factor determining message latency [5]. The idea is to try to serve a request with contiguous allocation, and to look for noncontiguous

additions only on demand. Under our noncontiguous scheme, a partial allocation is sustained, and the search wave continues only looking for the additional processors.

4 Performance Analysis

In order to investigate the potential and the feasibility of the proposed dynamic allocation in large PC clusters we conducted (i) fragmentation experiments and (ii) allocation overhead experiments. For (i) we used the Siemens hpcLine Primergy High Scalable Compute Sever at the Paderborn Center for Parallel Computing (PC^2). The Primergy Server is a distributed memory multicomputer with 96 compute nodes (two Intel Pentium II with 450 MHz and 512 megabyte DRAM) connected by a 500 megabyte per second unidirectional two dimensional SCI mesh [3], with wormhole XY routing. Programs were written in a special MPI version for the SCI hardware (ScaMPI) that run over the Solaris operating system. Our discrete event simulator 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 the generated allocation messages. It was used in (ii) to study the effects of fragmentation for the proposed strategies in bigger machines (up to 1024 nodes).

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 [5,9]. In these experiments, jobs arrive, are scheduled with a first-come, first-serve policy (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.

The strategies simulated in these experiments are a dynamic and a static version of the distributed noncontiguous free-form Leak algorithm, a static contiguous version of Leak and the contiguous structure preserving Frame Sliding [5]. Frame sliding 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. 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 the greater demands when jobs arrive faster than they can be processed. Jobs only delay for an exponentially distributed service time with mean of 10.0 time units. For example, under a system load of 1.0, jobs arrive as fast as they are serviced, on the 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 programs four processor profiles are randomly generated for each job: constant, increasing, decreasing and triangular. In the constant profile the processor demand do not vary during execution. By the increasing and decreasing profiles the processor demand varies from 1 to job size and from job size to 1 respectively during execution. The triangular 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 then decreasing to 1 again in the second half. For each job size distribution in these experiments, we measure: *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 completes, *System Utilization* (Su): the percentage of processors that are utilized over time and *Messages per allocation* (Mpa): the total number of generated messages by the processor management to allocate the incoming requests divided by the number of generated requests.

All simulations model a 32×32 mesh and run until 1,000 jobs have been completed. Results reported for the fragmentation experiments represent the statistical mean after 10 simulation runs with identical parameters, and given 95 percent confidence level, mean results have less than five percent error. Table 1 shows how well 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.

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

As expected, we can see that the dynamic strategy achieved the highest system utilization since it is the only strategy that can cope with the dynamic processor requests. Static strategies have to allocate fixed partitions with the highest number of needed processors increasing the internal fragmentation. It is important to notice that this not always results in the highest throughput and lowest job response time. Especially with high load, the optimistic approach of the dynamic allocation (no reservation are made for possible future increases

in the number of processors) may result in partitions that are allocated but do not have space to grow. The processing time of these partitions have to be extended, increasing the response time and reducing throughput. The dynamic strategy also profits from allocating free-form noncontiguous partitions. Bigger partitions are difficult to find in contiguous schemes resulting in long search waves and a lot of tries, each of them increasing the number of messages and time. In a noncontiguous scheme, allocations are cumulative resulting in shorter search waves and no waste of messages and time. The difficulties of structure-preserving contiguous allocation can be verified with the frame sliding strategy and the resulting poor system utilization.

Figure 3 (left) show the average job response times for the uniform job size distributions at varying system loads. For the uniform distribution, the system cannot maintain stability with the contiguous FS strategy past a system load of about 1.5. At this point, the job response times for this algorithm begin to increase very sharply. However, for the noncontiguous strategies, represented by Leak, with the uniform distribution, the system remains stable until a system load of about 2.5, where job response times begin to increase significantly, though not as dramatically as with the other strategies. Notice that the curves in these graphs begin to reach a plateau at high average response times. This is due to the fact that the simulated job stream is finite in length, and all response times are bounded by the overall finish time of the simulation. For an infinite job stream, the response time curves would continue to increase exponentially, resulting in near-infinite response times at high system loads. Figure 3 (right) graph the system utilization for these same algorithms and job size distribution at varying system loads. Notice that peak utilization is reached just after the same load where the system was seen to become unstable in its response time graph. All three strategies attained their peak utilization at system loads of about 3.0. The dynamic noncontiguous Leak reached up to 97 percent utilization, whereas the contiguous Leak reached only 65 percent because of the fixed partitions. The structure preserving contiguous leak was only able to reach 51 percent because of the high internal and external fragmentation.

The results measured in these experiments are all consistent with those reported by Zhu in [9] for the contiguous Frame Sliding strategy and by Lo [5] for the noncontiguous strategies. These fragmentation experiments indicate that dynamic noncontiguous allocation is superior to static noncontiguous allocation and far superior to contiguous allocation in terms of its ability to utilize the processors. Because noncontiguous allocation can always allocate a job if there are enough processors available, eliminating external fragmentation, it is shown to achieve higher system utilization. Thus, noncontiguous allocation allows for greater job throughput. However, these results ignore the increased communication contention that may be introduced as a result of noncontiguous allocation. This is not significant in machines with little contention like our Primergy Server or with switched machines like Myrinet clusters with no contention at all, but should be evaluated in machines where message contention could become a problem.

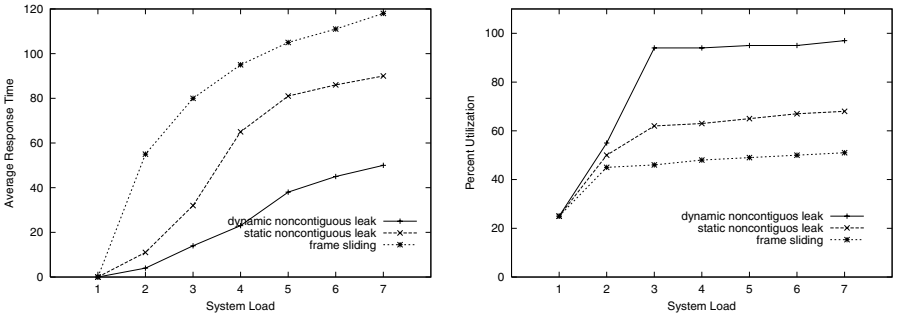

Fig. 3. Average job response time vs. system load and system utilization vs. system load for the uniform distribution of job sizes

Allocation Overhead As a first step in evaluating the feasibility of the dynamic allocation with current technologies we implemented the processor managers in the Primergy server and simulated incoming requests (the same load generation module of the simulator was used). Only 64 nodes were used for this experiments connected as an 8x8 torus. The incoming parallel jobs were not actually loaded in the machine and the allocated partitions are only reserved during the job duration and only global allocations are generated. In our preliminary performance test for a medium system load (5.0) we obtained allocation times around 0.03s for dynamic allocation and 0.162s for static allocation. 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 then in the static version. This results also in a reduction in the average time for an allocation. Altough, since in our test for each static allocation 10 dynamic operations are realized in mean (partition duration has a mean of 10) the total time needed for the dynamic allocation of all jobs is around twice as slow then the static allocation.

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

5 Conclusions

This paper proposes a dynamic distributed processor allocation strategy for parallel programming models that allow dynamic creation and deletion of tasks,

resulting in growing and shrinking processor demands. The dynamic strategy is built up on our distributed allocation model, in which the central entity responsible for processor status control is eliminated and the allocation operations are executed in parallel in the processor mesh itself. The basic allocation operations were redefined to match the characteristics of this new dynamic environment and implemented in a distributed processor manager. A distributed dynamic noncontiguous allocation strategy was evaluated and compared to static free-form and structure preserving in a mesh-connected 96-node multicomputer and results were simulated for bigger machines.

Our study shows that the dynamic distributed approach is feasible for large cluster machines with current communication technologies and permitted a greater parallelization of the allocation operations, eliminated the bottlenecks of the centralized model, and achieved a much better utilization of the processors. As a result, system utilization for the noncontiguous version of our dynamic algorithm reaches as high as 97 percent.

We conclude that distributed dynamic allocation provides a new approach that will help highly parallel systems to achieve better price/performance ratios in high demand, multi-user environments.

References

1. G.-M. Chiu and S. Kung Chen. An efficient submesh allocation scheme fot two-dimensional meshes with little overhead. *IEEE Transactions on Parallel and Distributed Systems*, pages 471–486, 1999. 785
2. H. Choo, S.-M. Yoo, and H. Y. Youn. Processor scheduling and allocation for 3d torus multicomputer systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 475–484, 2000. 785
3. IEEE. Ieee standard for scalable coherent interface (sci). *IEEE 1596-1992*, 1992. 784, 788
4. G. Kim and H. Yoon. On submesh allocation for mesh-connected multicomputers: A best-fit allocation and a virtual submesh allocation for fautly meshes. *IEEE Transactions on Parallel and Distributed Systems*, 9(2), feb 1998. 785
5. V. Lo and et al. Noncontiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):712–726, July 1997. 785, 787, 788, 790
6. V. K. Naik, S. K. Setia, and M. S. Squilante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Supercomputing 1993*, pages 824–833, 1993. 785
7. L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Transactions on Computers*, 1993. 787
8. C. A. F. D. Rose, H.-U. Heiss, and P. Navaux. Distributed processor allocation for large pc clusters. In *9th International Symposium on High Performance Distributed Computing*, 2000. 785, 787
9. Y. Zhu. Fast processor allocation and dynamic scheduling for mesh multicomputers. *International Journal of Computer Systems Science and Engineering*, 2(11):99–107, 1996. 785, 788, 790