Load Balancing For Network Based Multi-threaded Applications

Oliver Krone[‡], Martin Raab[¶⋆], Béat Hirsbrunner[‡]
[‡]Institut d'Informatique, Université de Fribourg, Switzerland
[¶]Institut für Informatik, Technische Universität München, Germany

Abstract. In this paper we present LBS, a load-management-system for network based concurrent computing. The system is built on PT-PVM, a library based on the PVM system. PT-PVM provides message passing and process management facilities at thread and process level for a cluster of workstations running the UNIX operating system.
The presented system is realized as an open library which can be easily used to implement new load-balancing algorithms. In addition to that, the unit of load which has to be distributed (either data or light-weight processes) can be transparently adapted to application needs. Therefore the system serves as an ideal test-bed for comparing different load-balancing methods.

1 Introduction

With the increasing computing power of dedicated workstations and the observation that an enormous potential of CPU cycles is merely unused because workstations tend to be idle most of the time, network-based concurrent computing (also commonly referred to as NOWs, networks of workstations) has recently become a major research topic. Other motivating aspects include heterogeneity, excellent price/performance characteristics and powerful development tools [11] for these machines.

The "usual" unit of parallelism for network based concurrent computing is a heavy-weight UNIX process, however in the past years a new type of process, called light-weight process or "thread" has emerged which has a couple of advantages compared to heavy-weight UNIX processes. Applications using this process type are commonly referred to as multi-threaded applications.

The notion of "thread" has been introduced as early as 1965 by Dijkstra [3] as a sequential flow of control, and is defined as *a sequence of instructions executed within the context of a process* [9]. It has its own program counter, register set and stack, but shares a common address space and operating system resources with other threads in a process. Multi-threaded applications provide many advantages for example) such as: small context switching time, fast thread to thread communication, increased application responsiveness due to overlapping of computation and communication [12].

The rest of this paper is organized as follows: in Section 2 we summarize some characteristics of load-management schemes in general and for NOWs in

⋆ partly supported by the German Science Foundation (DFG), grant Ma 870/5-1

particular, Section 3 is devoted to a detailed description of our system, Section 4 shows some preliminary performance results, and Section 5 concludes this paper and gives an outlook on future work.

2 Classification of Load Management Systems

A distributed application can be modeled by an *application graph* $H = (U, F)$ The nodes represent the application processes and their computational requests and the edges represent the communications and their intensity.

A distributed computer can also be modeled by a *hardware graph* $G = (V, E)$ where V is the set of processors and the edges represent the communication links between the processors. In this model, the load-management-problem can be stated as graph-embedding problem: we are looking for a function $\pi : U \rightarrow V$ which minimizes a given cost function.

Depending on the application graph used, one can distinguish the following two cases: (1) The application graph does not change over time. In this case π can be determined at application set up time and the load balancing functionality is reduced to an *embedding problem* or *mapping problem* depending whether a cost function is taken under consideration or not. (2) The application graph changes, which means that the mapping of load onto the available resources is done at runtime, and is adapted according to dynamic properties of the actual configuration, such as the load of a CPU, available network bandwidth and the like. In the sequel we will concentrate on this case, also known as the *dynamic load-balancing problem.*

A Model for Load Management Systems on NOWs A heterogeneous network of workstations is modeled by a vertex-weighted graph $G = (V, E; \alpha)$ where $\alpha : V \rightarrow [0, 1]$ is the normalized performance of a processor: Let t_{v_0} be the time needed by processor v_0 to execute a sequential process, and t_v the time needed on processor v, then $\alpha_v := t_{v_0}/t_v$ is the *performance* of v with respect to v_0. The *average load* $\overline{w_V}$ of a set V of processors, with w_v as the load of a processor $v \in V$, is defined by $\overline{w_V} := \frac{\sum_{v \in V} w_v}{\sum_{v \in V} \alpha_v}$, and the *optimal load* of a processor v is given by $w^*_v = \alpha_v \overline{w_V}$. The *speedup* is defined as usual as: $S(V) := t_{v_0}/t_V$ where t_V is the time needed by the processors in V, but now depends on the chosen reference machine v_0. In order to get a meaningful notion of *efficiency* one slightly changes the normal definition of efficiency: instead of dividing the speedup by the number of processors, one divides it by the total performance α_V of the network, where $\alpha_V := \sum_{v \in V} \alpha_V$. Note that the efficiency is independent of the machine of reference.

A similar model applies for *non-dedicated environments* where the available processor-performance may change due to other applications running on the same processor. The load induced by the processing of other applications can only be controlled by load-management systems working at operating-system level. Application-integrated load-management systems cannot influence the workload distribution of other applications. This load is thus referred to as *external load*, see for example [7] for an relationship of load average to program execution time on networks of workstations. *Internal load* is the load generated by the application and thus the load which can be manipulated by the load-management system.

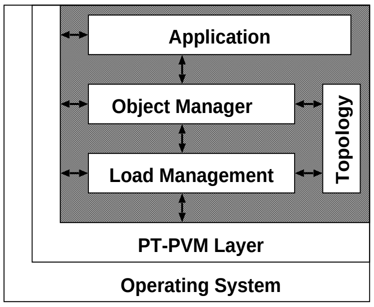
A load-management scheme must now determine the load itself (both external and internal), specify how it can be measured, and when and where it is created. There are several approaches for this in the literature, for example the gradient model [5] uses a threshold method to describe the load (low, middle, high), in [13] a quantitative approach is used, and in [2] a qualitative method is proposed.

Load-Balancing on Heterogeneous, Non-dedicated NOWs. After having studied several factors which influence the efficiency of load balancing systems (for a general overview see [6]), we identified the following important properties for load balancing schemes on heterogeneous, non-dedicated NOWs:

- Due to the heterogeneity at different levels (hard-and software), a load balancing scheme for NOWs should be *application integrated*. It seems impossible to adapt all the different machines used in NOWs for load balancing purposes at operating system level;
- Heterogeneity at hardware level implies that the performance with respect to a reference machine must be taken into consideration. Because these machines are typically used by other users (non-dedicated), their *external load* should influence the load balancing system;
- Communication on a LAN is a relatively expensive operation, compared to the high performance networks of "real" parallel computers, therefore the migration space of the load to be transferred is limited, that is, *locality aspects* like the neighborhood of a CPU should be considered;
- Node *information* (actual load and the like) may already be obsolete or *outdated* when the information arrives at another node, a load balancing system for NOWs should consider this;
- NOWs are typically connected via a bus and do not use a special topology, such as a grid for example. Since many load balancing methods in the literature assume that the CPUs are connected in a regular topology, a load balancing system for NOWs must provide means to arrange the workstations in a (virtual) *topology*;
- Load balancing schemes for NOWs are *asynchronous*, a synchronous variant would introduce too much administrative overhead.

3 LBS: Load Balancing System

This Section introduces LBS *(Load Balancing System)* [8]. The main goal was to combine the advantages of operating-system integrated load-balancing tools (*i.e.* application independence, ease of use, etc.) with those of application-integrated load-balancing libraries (*i.e.* performance) and to provide an easily extensible load-management tool which may serve as a testbed for different load-balancing algorithms. In order to achieve this, LBS was built in a modular way, such that new load-balancing algorithms may be easily integrated into an existing system without changing the application program.



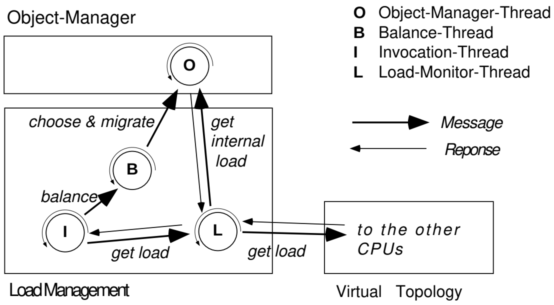


Fig. 1: Basic Structure of Lbs. **Fig. 2:** Internal Structure of Lbs.

Basic Structure of Lbs. From the operating-system's point of view, Lbs is a part of the application program because the load-balancing is done at application-level, whereas from the application programmer's point of view, Lbs is simply a library which is linked to the application.

Lbs is built on Pt-Pvm [4] which provides a Pvm [10] like programming model at thread-level. All the communication between the application threads on the different machines is done using Pt-Pvm. Fig. 1 illustrates the interplay between the different software components of Lbs.

Lbs does not know the type of the load objects. Therefore the interface between Lbs and the application program must be implemented by a component called `Object Manager`. The rest of the Lbs system communicates with the `Object Manager` via well-defined functions which allow to evaluate the current load and to move load-objects between neighborhood machines.

The `Load Management` comprehends the functionality of a control feedback control system and is described in detail in the following Section.

Most of the load-balancing algorithms try to achieve system-wide load-balance by local load movements. Thus we need a notion of "locality", *i.e.* a library which implements a topology. This is done by the (virtual) `Topology` component in Lbs.

Internal Structure of Lbs. The internal structure is illustrated in Fig. 2. The structure is influenced by the following approach: In order to emphasize the load balancing process in time, one models the load-balancing problem as a feedback control system and splits it up into three phases:

– During the *load* capture phase, the load-balancing system collects load-information of the neighbor machines and activates, if necessary, the decision-phase, otherwise the load capture component falls asleep for a time determined by the load-balancing algorithm and then starts over again.
– In the *decision*-phase the load-balancing system decides *whether* load-balancing has to be done at all and if so, determines *how much* load has to be moved and determines the *sender* and *receiver*. Depending on the type of the load, the system also has to determine *which* load-objects have to be moved.

```
void farmer(FARMER_PARAM prm) {
  TASK task; RESULT res, size_t size;
  forall tasks
    distribute_task(prm, &task, sizeof(task));
  forall results
    get_result(prm, &res, &size);
  compute_final_result();
}
```

```
typedef struct { ... } TASK;
typedef struct { ... } RESULT;
void worker(WK_PARAM prm,
  void *task, size_t size) {
  RESULT res;
  res = compute_result();
  return_result(prm, &res, sizeof(res));
}
```

Fig. 3: TfLbs application skeleton.

– During the *load-migration* phase the actual migration takes place.

These three phases have a strong influence on the design of most load-balancing systems and lead to a modular structure of the system. In the case of Lbs, the different phases are implemented by Pt-Pvm threads:

– The `load_monitor_thread` implements the load capture component. This thread is responsible for getting the internal and external load and for getting the load of the neighbors. Because the type of the load-objects is not known to Lbs, this is done by communicating with the `object_manager_thread`. The external load is evaluated by calling operating system services.
– The `invocation_thread` implements the invocation policy. Depending on the chosen policy, the `load_monitor_thread` is queried for the current load, and if necessary load-balancing is activated by sending a request to the `balance_thread`.
– The `balance_thread` does the actual load-balancing. When receiving a load-balancing request, the load-movements are computed and the object-manager is invoked to move the load in the appropriate directions.

TfLbs— **Load-balancing for Task-Farming.** Many problems may be formulated using the *task-farming* paradigm: A task-farming program consists of a number of independent subproblems, which may be solved independently on any processor. Data dependencies between tasks can be handled by creating the tasks only when all the tasks from whom information is needed have already been solved.

The TfLbs library provides a simple interface for task-farming programs and handles the interaction with the Lbs system. It is thus an instance of an object-manager. TfLbs application programs consist of a main-function called `farmer()` which creates the tasks (by calling `distribute_task()`) and collects the results (`get_result()`) at the end of the computation.

The tasks are computed by a `worker()`-function. `worker()` gets a parameter of the type `WK_PARAM` which is application-dependent and allows the programmer to identify the tasks. During the execution of a task, new tasks may be created and at the end the result is sent to the farmer by calling `return_result()`. The outline of a typical TfLbs application is sketched in Fig. 3.

The actual implementation of TfLbs is as follows: on each machine in the Lbs system an `object_manager` is installed which handles the list of tasks assigned to that machine. Another thread called `worker_thread` fetches waiting tasks from the task-queue, processes them and sends the results back to the farmer. The load-balancing is done in the background by moving tasks from longer queues to empty queues.

The implementation of the `object_manager` is thus quite simple: the `object-manager` thread must handle the queue of tasks assigned to a machine. The queue management supports the following operations:

- Returning the length of the queue in order to estimate the internal load of the machine;
- Inserting new tasks into the queue;
- Removing a task from the queue: the `object_manager` supports two ways of removing tasks from the queue: the *synchronous* remove and the *asynchronous* remove. When the `worker_thread` asks the `object_manager` for a new task, and the queue is empty, the `worker_thread` has to block until some new tasks arrive, whereas it is possible that Lbs might want to reallocate more tasks from one machine to another than there are currently available — in this case Lbs must not block, it is sufficient to inform Lbs that there are no more tasks on the machine.

The other crucial part of TfLbs is the `distribute_task()` function. It is the function that assigns the first tasks to the machines. The more efficiently this is done, the less load-balancing is needed. One possible approach is to assign the new tasks always to the neighbor with the least load. Another possibility is to assign the tasks to a randomly chosen machine, hoping that this leads to a more or less balanced load situation — it has been shown that the maximal load can be reduced exponentially by randomly choosing two processors and assigning the new task to the least loaded [1].

4 Performance Results

We have tested Lbs with a simple program that computes the Mandelbrot set. This program is based on the TfLbs library. The Mandelbrot program splits the area for which the Mandelbrot-set is computed into $M \times N$ rectangles each containing $m \times n$ points. Each of the rectangles is computed by a different task. The running-time distribution is thus given by the number of points contained in each rectangle and by the "depth" of the computation, *i.e.* the number of iterations considered when determining the convergence.

For this experiment we implemented the Local Pre-computation-based Load Balancing Algorithm (LPLB), a variant of the PLB-algorithm [2], in Lbs and used a load-sharing calling strategy, *i.e.* every time a PE[1]runs out of work, the load will be locally re-balanced. Fig. 4 and 5 show the distribution of load on 6 Sparc stations 5 running Solaris 2.5, PVM 3.3.9 and Pt-Pvm 1.13 with and without Lbs, respectively.

[1] Process Environment, a UNIX process which may host one or more Pt-Pvm threads.

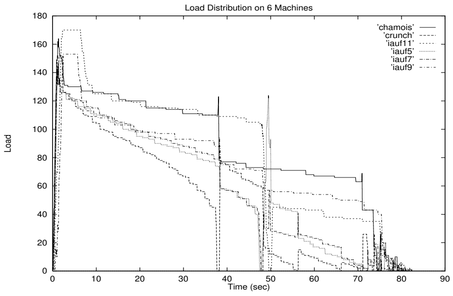


Fig. 4: Distribution of Load using LBS.

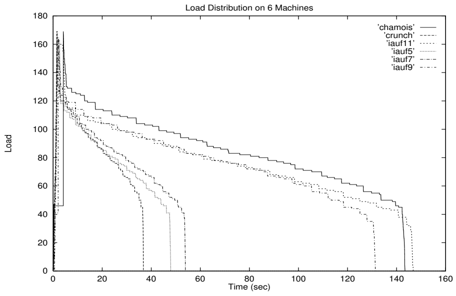


Fig. 5: Distribution of Load without LBS.

Table 1 shows the average execution time in seconds for several test sets and different configurations of LBS. The test sets consist all of $M \times M$ tasks, each containing $m \times m$ points. The "depth" of the computation was the same for all test sets. For "lb" the LPLB load-balancing-algorithm and the load-sharing calling strategy is used. In "lb_noex" the same algorithm is used, but the external load is not considered, "lb_noex_ABKU" uses the mapping-algorithm described in [1]. In "no_lb" and "nolb_noex_ABKU" no load-balancing is done at all. The results show that LBS significantly reduces the the running time for NOWs that consist of more than 2 PEs, whereas if only two PEs are used, the overhead due to LBS dominates. One also sees, that the ABKU-mapping algorithm doesn't behave well in our case, which is due to the fact that all the load is created at the same time.

5 Conclusion

This paper described LBS, a system for load balancing for network based multi-threaded applications. Starting from a classification of design issues for load-management systems, we identified significant properties for load-management systems to be used for NOWs. These recommendations have been realized in our system, LBS, which is implemented on top of PT-PVM, a software package based on the PVM system which provides message passing and process management facilities at thread and process level for a cluster of workstations running the UNIX operating system. LBS's distinct features include a modular multi-threaded software structure, its object-, load-, topology-, management is implemented as separate PT-PVM threads. By implementing a special object manager in the form of the TFLBS library, we showed how the efficiency of a well known parallel programming model (task-farming) can be increased if used in combination with a load-management system for concurrent network based computing.

Future work include a dynamic virtual topology management (for the moment the topology is fixed at application startup), and an improved mechanism to determine the external load of a workstation.

In addition to that, we are currently implementing new object managers, for example object managers which change the topology dynamically according to the actual load in the system, and started some promising experiments with threads as load objects.

		$M = 16$			$M = 32$		$M = 64$
# PEs	Configuration	$m = 16$	$m = 32$	$m = 48$	$m = 32$	$m = 64$	$m = 64$
1	nolb_noex	57.77	248.44	553.56	885.15	2738.19	
2	lb	33.67	137.16	299.10	535.46	2095.07	8401.88
	lb_noex	33.32	134.24	292.04	538.01	2096.38	7553.68
	lb_noex_ABKU	33.02	142.10	303.53	537.89	2125.26	7938.18
	nolb	31.41	131.28	291.50	516.11	2056.58	7992.80
	nolb_noex_ABKU	31.52	147.78	323.92	516.66	2058.75	7568.18
4	lb	10.71	34.55	73.59	131.17	483.59	2006.93
	lb_noex	10.54	33.47	71.76	132.40	486.65	1969.86
	lb_noex_ABKU	10.90	37.95	76.06	132.39	503.26	1998.43
	nolb	12.10	40.74	87.71	159.37	597.68	2552.75
	nolb_noex_ABKU	12.39	48.59	106.96	162.15	676.49	2370.51
8	lb	8.63	22.19	46.68	70.95	271.11	1003.34
	lb_noex	8.21	19.72	41.07	70.63	258.05	1000.05
	lb_noex_ABKU	8.82	25.34	43.42	72.42	279.38	1003.11
	nolb	12.00	32.24	71.92	129.68	494.68	2296.43
	nolb_noex_ABKU	12.32	34.25	74.70	131.57	553.18	2320.82
12	lb	8.15	15.04	30.11	54.59	180.04	683.60
	lb_noex	8.26	14.76	31.37	52.56	181.61	684.13
	lb_noex_ABKU	17.55	20.56	33.47	89.40	193.59	756.86
	nolb	8.53	21.88	44.29	94.74	330.67	1342.02
	nolb_noex_ABKU	15.61	24.10	54.07	97.63	348.71	1383.19

Table 1: Average execution times.

References

1. Y. Azar, A.Z. Broder, and A.R. Karlin. On-line load balancing (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science*, pages 218–225, Pittsburgh, Pennsylvania, 24–27 October 1992. IEEE.
2. M. Böhm. *Verteilte Lösung harter Probleme: Schneller Lastausgleich*. PhD thesis, Universität zu Köln, 1996.
3. E. W. Dijkstra. Cooperating sequential processes. *Programming Languages*, 1965.
4. O. Krone, M. Aguilar, and B. Hirsbrunner. PT-PVM: Using PVM in a multi-threaded environment. In 2^{nd} *PVM EuroPVM '95*, Lyon, France, 1995.
5. F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, January 1987.
6. T. Ludwig. *Lastverwaltungsverfahren für Mehrprozessorsysteme mit verteiltem Speicher*. PhD thesis, Technische Universität München, München, Dezember 1992.
7. T. E. Meyer, J. A. Davis, and J. L. Davidson. Analysis of Load Average and its Relationship to Program Run Time on Networks of Workstations. *Journal of Parallel and Distributed Computing*, 44(2):141–146, August 1997.
8. M. Raab. Entwicklung eines Lastverwaltungssystems für vernetzte Arbeitsplatzrechner. Master's thesis, University of Fribourg, 1997.
9. Sun Microsystems, Mountain View, California. *SunOS 5.3 Guide to Multithread Programming*, November 1993.
10. V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.
11. S. White, A. Alund, and V.S. Sunderam. Performance of the NAS Parallel Benchmarks on PVM-Based Networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, 1995.
12. N. Wirth. Tasks versus Threads: An Alternative Multiprocessing Paradigm. *Software – Concepts and Tools*, (17):6–12, 1996.
13. C. Xu, B. Monien, R. Lüling, and F. Lau. An analytical comparison of nearest neighbor algorithms for load balancing in parallel computers. In *Proc. of the 9th International Parallel Processing Symposium (IPPS '95)*, April 1995.