A General Architecture for Load Balancing in a Distributed-Memory Environment

Hiroshi Nishikawa

Tokyo Information Systems Research Lab.
Matsushita Electric Industrial Co., Ltd
4-5-15, HigashiShinagawa, Shinagawa-ku, Tokyo 140

Peter Steenkiste

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

The goal of load balancing is to assign to each node a number of tasks proportional to its performance. On distributed-memory machines, it is important to take data dependencies into account when distributing tasks, since they have a big impact on the communication requirements of the distributed application. Many load balancers have been proposed that deal with applications with homogeneous tasks, but applications with heterogeneous tasks have proven to be far more complex to handle. In this paper we present a load balancing architecture that can deal with applications with heterogeneous tasks. The idea is to provide a set of load balancers that are effective for different types of homogeneous tasks, and to allow users to combine these load balancers for applications with heterogeneous tasks. We implemented this architecture on the Nectar multicomputer and we present performance results for several applications with homogeneous and heterogeneous tasks.

Keywords: load balancing, heterogeneous, distributed-memory

1 Introduction

Distributed-memory multiprocessor are becoming the application of choice for computation-intensive applications, because they are scalable. However, they are much harder to program than sequential or shared-memory machines. To achieve good performance on distributed memory systems, programmers need a programming model that is suitable for exploiting various kinds of parallelism, and support for load balancing so that work is assigned to each node according to its performance. In this paper, we focus on the issue of load balancing.

Load balancing can be static or dynamic. With static load balancing, the task and data distribution is determined at compile time. The distribution can be done by the compiler, possibly with input from the programmer [7, 16, 18]. Unfortunately, applications can often not be partitioned optimally at compile time because their behavior is data-dependent. Furthermore, if the application is distributed over a set of computer systems connected by a high-performance communication network [1], static load balancing is ineffective, because nodes are often shared with other users, making their load unpredictable. On this type of distributed system dynamic load balancing is indispensable.

With dynamic load balancing, work is assigned to nodes at runtime, and information about the status of the nodes and the application can be used to optimize the assignment. There is a lot of theoretical work on load balancing [3], but only a small number of relatively simple load balancing algorithms have been used in practice [8, 15, 6, 12, 11, 17, 19, 5]. These load balancers have typically been optimized for a certain class of applications, for example, applications without data dependencies between the tasks.

Most load balancers were designed to handle applications with homogeneous tasks, for example, data-parallel applications or tree-based algorithms. A lot of applications however consist of heterogeneous tasks, i.e. tasks performing different operations or operating on different types of data. An example is a producer-consumer application where one type of tasks generates data that is consumed by a different type of tasks. Load balancing for heterogeneous applications is harder because different tasks have different costs, and the data dependencies between the tasks can be very complex. It is important to consider data dependencies when doing load balancing, because the placement of the tasks, relative to where the data it uses, strongly affects the communication overhead and thus the overall performance of the application [13].

We propose a dynamic load balancing architecture that applies to a wide range of applications, including applications with heterogeneous tasks. Our system several several load balancers that have proven to be effective for homogeneous tasks. For applications with heterogeneous tasks, it provides a load balancing framework that is built on top of the homogeneous load balancers. The framework consists of a database that describes the progress of each of the nodes, primitives to redistribute work, and a carefully designed "upcall" interface that is used to invoke the application-specific load balancer when there are changes in the runtime environment. The load balancing framework simplifies the task of writing a load balancer since the programmer can concentrate on the essential tasks: when and where to move work. Collecting information about the environment and progress on the compute nodes and moving work, are handled by the system.

The paper is organized as follows. Section 2 reviews the related work. In Section 3 we briefly describe the programming model in which we implemented our load balancing framework and in Section 4 we present our load balancer. Section 5 shows examples of load balancing of both homogeneous and heterogeneous applications, and presents performance results for four applications. Finally, we summarize our results in Section 6.

2 Previous Work

Implemented load balancing methods fall in three categories. In the first group, there is conceptually a system-wide global task queue from which tasks are assigned to idle processors. Methods differ in how many tasks are assigned at a time. In chunk scheduling (or self scheduling) [8], a constant number of tasks is scheduled at a time. In guided self scheduling (GSS) [15], the number is proportional to number of remaining tasks in the queue. Factoring [6] takes into account both the number of tasks in the queue and the number of processors used by the application. Tapering[12] selects a chunk size based on the distribution of the observed task execution time. These methods have mainly been implemented on the shared memory multiprocessors; only tapering has been implemented on a distributed memory system.

The second group of load balancers takes into account the hop length between nodes, so they are typically used on distributed-memory machines with a regular interconnect (i.e. torus). In the gradient model [11], a "node load potential" field is established over the nodes, and newly created tasks drift to the node with the global minimum value through the potential field. ACWN [17] restricts the hop length, so that tasks are assigned to nearby nodes. The hop length is determined by the load on the nodes. Adaptive load sharing [19] uses a central manager that gathers node information periodically and redistributes tasks if needed. The period is inverse-proportional to the load imbalance in the system. Load balancers in this group do not take the data/task affinity into account, so they are mainly applicable to DOALL-style applications.

Finally, an important class of load balancing methods is based on data decomposition [5]. With this method, data structures are partitioned across the nodes, and each node operates on the data that is assigned to it. How much data is assigned to a node depends on its computational resources. When performing load balancing for one iteration of the loop, the elapsed time on each node for the previous iteration of the loop can be used to redistribute the data. This method makes it possible to minimize remote data references by taking data dependencies into account during data repartioning.

The above load balancers are effective on some classes of applications, mainly homogeneous applications with specific types of data dependencies. Load balancing for heterogeneous tasks is more complicated because the data dependencies between the heterogeneous tasks can be very complex, and it is too hard for a load balancing system to take these dependencies (and corresponding data locality) into account when performing load balancing in a distributed-memory environment. The load balancing architecture described in the next section addresses this issue by providing an environment that supports the development of application-specific load balancers.

3 Base Programming Model

Before we discuss the general load balancing architecture, we describe Aroma [14], our distributed programming system. Aroma programs consist of tasks that operate on distributed objects. Tasks are segments of sequential code: once the task starts it will run until completion without blocking. Tasks can read in one or more shared objects when they start, and write objects upon completion. Tasks are created dynamically from an initial main program and from other task.

Aroma adopts weak consistency [4] to reduce the overhead associated with synchronizing updates to data in a distributed-memory environment. Aroma only explicitly serializes access to data objects if the programmer indicates that this is necessary. Data objects also support task synchronization. A task can for example block on a data object of type iterator, waiting for a new copy of the data to be written; it becomes runnable as soon as the object is written. Aroma supports both task and data parallelism. Data parallelism is supported through aggregate objects that are distributed across the compute nodes, similar to the data partitioning performed by parallelizing compilers.

One of the most critical features of a distributed programming system is the ability to hide the latency of remote data accesses. Aroma hides latency by prefetching the input objects of tasks. If an object is located on a remote node, the task scheduler executes other tasks while the data is being fetched, and only after all data is available is the task placed on the run queue.

Load balancing in Aroma can be done statically or dynamically. The programmers can control load balancing by specifying the location of tasks and shared data using Aroma directives. Dynamic load balancing is done by automatically migrating shared data and tasks based on runtime information, as is described below.

4 A General Load Balancing System

We describe the overall architecture of a load balancer that supports both applications consisting of homogeneous and of heterogeneous tasks.

4.1 Homogeneous Tasks

Many load balancers have been defined for homogeneous systems. In this section we described three algorithms, each suitable for a different type of task. These load balancing policies have been implemented in Aroma and performance results are presented in Section 5.

The **loose factor-based policy** is effective for balancing tasks without dependencies, as illustrated in Figure 1-(a). It is an extension of the factoring policy

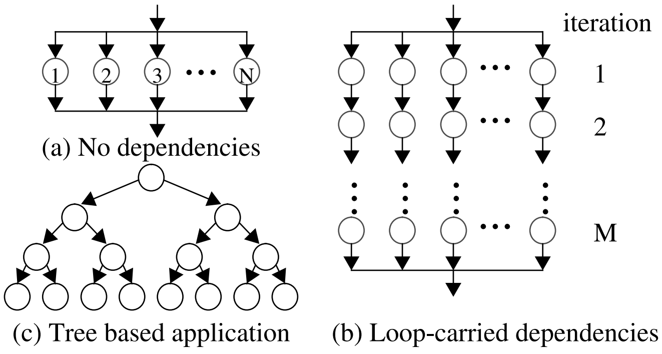

Figure 1: Tasks with different types of dependencies

described in [6] for distributed-memory architectures. Since there are no dependencies between the tasks, they can be executed and migrated without concern for communication or synchronization requirements. The tasks assigned to each node are placed in groups, and each node sends a status message to the load balancer after finishing the execution of all tasks in a group. Based on this information the load balancer can determine the the relative speed of the nodes, and it can issue instructions to move tasks from slow nodes to faster nodes if needed.

The size of the groups decreases as the computation progresses. Specifically:

```
Group[i+1] = Group[i] / 2
```

For example, when four nodes executes 1,000 tasks in total, each node will initially receive 250 tasks. The sizes of the groups on each node will be:

i	0	1	2	3	4	5	6	7
Group	125	62	31	16	8	4	2	2

As a result, a node will contact the load balancer after it has executed 50% of the remaining work, so the load balancing becomes more fine grain near the end of the computation, when fewer tasks are left.

The **phase-based policy** is used for balancing parallel iterative tasks as shown in Figure 1-(b). It consists of a distributed loop inside an outerloop, and there can be dependencies between tasks in successive iterations of the outerloop. This type of iterative task can be scheduled as follows. Each task receives an iteration number that corresponds to the loop index of the outerloop. All iterative tasks with the same (initial) iteration number are executed in parallel, and they spawn the corresponding tasks with the next iteration number when they finish execution. Each node executes the tasks with the lowest iteration number first.

The progress of nodes is measured using iteration numbers, similar to the way group numbers are used in factor-based load balancing. Nodes send their iteration number to the central load balancer when they finished execution of the last task with a given iteration number; this iteration number is called the phase value of the node. Since tasks are scheduled based on their iteration number, the phase value summarizes the progress of the node, and the load balancer issues migration instructions if the difference in phase values between nodes becomes too large.

With the **priority-based policy**, each task is assigned a priority when it is created. After a node executes a task, a status update message is sent to the central load balancer, which can assign the next task to that node. Priority-based load balancing can be used for a variety of tasks, including tasks with irregular behavior. An example is a tree search application as shown in Figure 1-(c). The priority can for example be the tree depth (breadth first search) or the negative of the tree depth (depth first search). Priority-bases scheduling is very flexibly, but it also introduces more overhead than the earlier policies since the compute nodes and the load balancer interact for every task, instead of for groups of tasks.

These three system-defined load balancers cover a wide range of applications with homogeneous tasks. We now discuss how these, or similar policies, can be used as building blocks in load balancers for heterogeneous tasks.

4.2 Dealing with Heterogeneous Task

When performing load balancing on a set of heterogeneous tasks it is attractive to use for each type of task a load balancer that is appropriate for that type, but this approach does not deal adequately with load balancing across task types. Moreover, having multiple independent load balancers can easily create problems. For example, multiple load balancers could react to an imbalance, resulting in too much work being moved, or a temporary imbalance caused by one load balancer could trigger undesirable work movement by other load balancers.

These problems can be avoided by having work movement controlled by a single entity that has a full picture of the activities in the entire application, and that also understands the dependencies between the different types of tasks. This global load balancer can however benefit from the presence of simpler load balancers that operate on tasks of a specific type. For example, when the system is near equilibrium, it will often be possible to do load balancing by migrating tasks of a single type, under control of a load balancer optimized for that task type.

These observations motivate our global load balancing architecture: it includes both a set of simple load balancing strategies for each of the task types in the application, and an application-specific global load balancer that manages the interaction between the different task types and their load balancers.

Figure 2 shows the system organization of the load balancer. The system consists of a set of compute nodes and a central task scheduler. The task scheduler collects status information from the nodes and issues task migration instructions based on this information. The task scheduler supports the three load balancing policies for homogeneous tasks described in the previous section. For more complex applications, the user has to provide an application-specific load balancer.

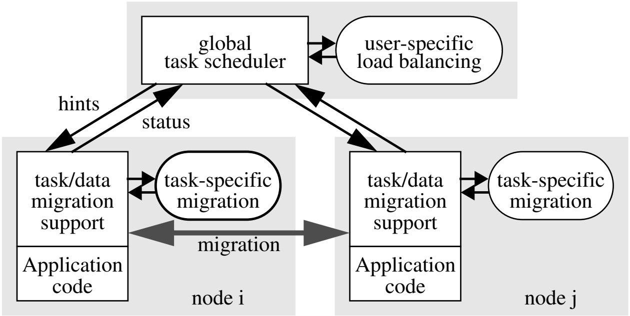

Figure 2: Aroma Load Balancer Architecture

This load balancer is responsible for the overall load balancing strategy, but it can use the simple system provided load balancers to manage each type of task.

A compute nodes include, besides the application code, a local task queue and scheduler, and code to send status information and to receive and handle task migration instructions. Migrated tasks are moved directly between the nodes to minimize overhead. Task-specific migration code deals with selecting the appropriate tasks, packing and unpacking state information and inserting migrated tasks in the appropriate place in the local task queue.

4.3 Load balancing operation

At start up time the user specifies a policy for each policy using the **balance_load** call. The effect is that the compute nodes will schedule the tasks following the rules that apply to that load balancing policy and will also send appropriate status updates to the global task scheduler. At runtime, the global task scheduler combines all the status information of all the nodes and task types in a database. This database is based on the **Status** structure and includes for example for tasks that follow phase-based load balancing, the phase value, average execution time per task, and the number of tasks per node.

The user also specifies for each task type an entry point into the application-specific load balancer using the primitive **define_load_balancer**. This function is called from the global task scheduler when the status information that applies to that type of task changes. This "upcall" mechanism allows programmers to perform efficient task scheduling for heterogeneous tasks. The user-specific load balancer can use the function **get_task_info** to retrieve the node status from the database, and can then make load balancing decisions based on that information. For example, it can decide that redistribution of tasks of a specific type is needed to achieve good performance, or that a node that was previously reserved for a single task type should now also execute tasks of different types.

If the load balancer decides that work has to be redistributed, it must construct a **Hint** structure. For the phase-based policy, this structure includes the source and destination node number, the phase numbers and the number of tasks that should be migrated. The **send_migration_hint** primitive will deliver the instructions to the computing nodes. For the priority-based policy, it can select the next task to be scheduled using the **next_priority** call.

The computing nodes handle migration instructions in the same way as for homogeneous applications (Figure 2). Which tasks should be migrated is determined by the task-specific function **get_task**. Tasks and the associated data are packaged for transmission and appended to a migration list using the functions **append_task** and **append_data** and the command **migrate** sends the tasks and data to the appropriate nodes. Upon arrival, the migrated tasks are processed by task-specific migration code (Figure 2). For example in the case of the phase-based policy, the tasks are executed until they reach the same phase as the local tasks before they are placed in the local task queue. These task-specific functions are declared using the functions **define_ship_in** and **define_ship_out**.

5 Examples and Performance

In this section we present examples and performance results for the dynamic load balancing architecture described in this paper. The performance is evaluated using three measures: 1/ the overhead of the dynamic load balancer on a static homogeneous system where no load balancing is needed, 2/ by comparing the performance of the load balancer in a heterogeneous system with the estimated optimal performance and 3/ by evaluating the task distribution across the nodes. The homogeneous node configuration consists of dedicated SUN4/330 nodes, and the heterogeneous configuration consists of dedicated SUN4/330 nodes and one dedicated SUN4/110 node. All nodes are connected using Nectar [1]. We present results for both homogeneous and heterogeneous applications.

5.1 Homogeneous Task Load Balancing

We show results for two applications that use the factor-based policy, and one application that uses phase-based load balancing. The size of the tasks is uniform in the first and third example, but grows monotonically in the second example. The implementation of the second and third example using Aroma is explained in detail in [14].

Matrix Multiplication(mm): The input objects for matrix multiplication are a matrix of row partitioned vectors and a replicated matrix, while the result matrix is partitioned by row. All tasks have the same size and there are no dependencies, so we use factor-based load balancing. We present results for a 256×256 matrix.

Prime Sieve(prime): A number is prime if it cannot be divided by any prime smaller than its square root. A parallel version of prime sieve is obtained by dividing the range of integers into segments, and have parallel tasks check whether the integers in each segment are prime or not. The initial set of prime numbers is computed before the parallel tasks start execution, and is replicated across all nodes. Since the number of primes that have to be used for the prime check increases as the integers become larger, the size of the prime check task increases, so the task distribution is non-uniform. There are no data dependencies, so factor-based load balancing can be applied.

nodes	seq.	1	2	3	4	5
mm	29.76	29.77	15.08	10.16	7.62	6.15
	-	29.77	15.03	10.34	7.87	6.42
prime	68.59	68.38	34.63	23.42	17.67	14.19
	-	68.57	34.45	23.35	17.68	14.30
poiss	11.87	13.29	6.58	4.45	3.63	3.02
	-	13.35	6.67	4.48	3.93	3.40

Table 1: Execution time in seconds in homogeneous environment, both without (first line) and with (second line) dynamic load balancing

#nodes		2	3	4	5
mm	measured	23.39	14.86	9.57	7.58
	optimal	22.91	13.00	9.06	6.96
prime	measured	46.90	29.43	21.93	16.02
	optimal	46.53	27.76	19.76	15.34
poiss	measured	11.03	7.52	4.59	4.19
	optimal	9.58	5.31	3.67	2.81

(a) measured and optimal time

	Sun4/330	Sun4/110
mm	29.76	97.0
prime	68.59	145.5
poisson	11.87	49.9

(b) execution time of sequential code

Table 2: Measured and estimated optimal execution time in heterogeneous environment

In the experiment, the segment size is 2,000 and the integer range is 1 to 1,000,000.

Poisson Equation(poiss): We use the odd-even successive overrelaxation method, and both the odd and even matrix are partitioned by row. Each task reads three consecutive vectors, generates the next version for the middle vector, and then spawns the task that will compute the next iteration. We use phase-based load balancing. In the experiment, the matrix size is 202×202 and the iteration number is 100. Since there are no tasks for the boundary rows, the total number of task executed is 20,000.

Table 1 shows the overhead of the system-defined load balancing policies on a homogeneous system. The overhead is relatively low in each case. Overall, the overhead is highest for **poisson**, which has the smallest task granularity. Increasing the number of nodes, while keeping the problem size fixed, has a different influence on the number of status messages used by the factoring and phase-based load balancers. The number of messages sent by each compute node remains constant for phase-based load balancing, but decreases for factoring, since the number of tasks assigned to a node decreases. This explains why, as the number of nodes increases, the overhead increases more rapidly for **poisson** than for **mm** and **prime**.

Table 2 shows the performance of the load balancer when one node is slower, so our load balancer has to move tasks early on in the computation. To evaluate the performance, the measured results are compared with the estimated optimal execution time calculated using the sequential times. The optimal times assume no parallelization, communication and load balancing overhead. The execution time of sequential programs are listed in the Table 2-(b). The measured time for **mm** and **prime** is fairly close to the estimated optimal time. The results for **poisson** are not so good, because it exploits relatively fine grain parallelism and requires a lot of communication; these problems become worse as the number of nodes increases.

type (Sun4)		110	330	330	330
mm	measured	26	78	77	75
	optimal	25	77	77	77
prime	measured	83	149	143	125
	optimal	68	144	144	144
poiss	measured	1488	6423	6230	5859
	optimal	1487	6171	6171	6171

(a) 4 nodes

type (Sun4)		110	330	330	330	330
mm	measured	21	64	60	60	51
	optimal	19	59	59	59	59
prime	measured	62	113	113	109	103
	optimal	52	112	112	112	112
poisson	measured	539	5685	4691	4682	4393
	optimal	1136	4716	4716	4716	4716

(b) 5 nodes

Table 3: Measured and optimal task distribution in heterogeneous environment

Another way of measuring the load balancing performance is to show how, after dynamic load balancing, the number of tasks assigned to each node matches the performance of that node. Table 3 compares the measured and the estimated optimal distribution of tasks using one Sun4/110 and three Sun4/330, and Sun4/110 and four Sun4/330. We again see a good match for **mm** and **prime**, but a larger difference for **poisson**, again as a result of a higher communication overhead.

5.2 Heterogeneous Task Load Balancing

Los Angeles Air Pollution is an application with heterogeneous tasks that follow a producer-consumer model. A set of producer tasks computes the wind velocity for every hour at each point of a regular grid over the area that is being simulated, based on measurement from weather stations and prerecorded weights. Based on this data, consumer tasks track particles released in initial locations of interest as they move about the grid. Previous reports [2, 9] describe the problem in more detail. In [9] the effect of static load balancing is shown. In [2], we use dynamic load balancing, but only the particle tracking tasks can be moved.

In this section we describe how our load balanc-

```
main()
{
  /* Initialization */
  balance_load(tracer, PHASE(time));
       /* time field is used as phase value */
  balance_load(comp_vel, PRIORITY);
  define_load_balancer(tracer,tracer_sch);
  define_load_balancer(comp_vel,comp_vel_sch);
  define_ship_in(tracer, tracer_ship_in);
  define_ship_out(tracer, tracer_ship_out);
  /* task startup code follows */
}
```

Figure 3: registration of LA load balancer

```
ship_in_tracer(t)
task *t;
{
  int cur_sim_time = local_phase(tracer);
  while (task_elem(tracer,t,time)<cur_sim_time)
    execute_here(t);
}

ship_out_tracer(hint)
Hint hint;
{
  task *tr;
  while (hint->num--> 0) {
    if ((tr = get_task(tracer)) != NULL)
      append_task(tr);
  }
  migrate_task(hint->d_node);
}
```

Figure 4: Task migration code of LA application

ing system can be used to build a dynamic load balancer that handles both types of tasks. Tracer tasks (**tracer**) calculate the next particle position from the current particle position and the two consecutive wind velocity data, and then create a new **tracer** task to calculate the next time step. Their behavior matches phase-based load balancing and the simulated time can be used as the phase value. The wind velocity tasks (**comp_vel**) are computationally expensive and are scheduled based on priority; the simulated time is used as the priority.

The application specific load balancer has two entry points: **comp_vel_sch** for the wind velocity tasks, and **tracer_sch** for the tracer task. Both schedulers are registered as shown in Figure 3. The ship_in and ship_out functions of the tracer task are also registered. Their operation is shown in the form of pseudo code in Figure 4. In **ship_in_tracer**, a migrated task is executed until it has the same phase value as the local tracer tasks. The **local_phase** primitive returns the phase value on the computing node. To get the time value of the migrated task, **task_elem** accesses

```
comp_vel_sch(node)
node_t node;
{
  Status nodes[MAX_NODES];
  get_task_info(tracer, nodes);
  for i=0 to max_node do
    min_con = min_con>nodes[i].phase?
              nodes[i].phase:min_con;
  next_pro = next_priority(comp_vel);
  if (next_pro - min_con >= HIGH_MARK)
    /* comp_vel executes too fast:
     * do not schedule a new one */
    return;
  else schedule_task(comp_vel, node);
}

tracer_sch(node)
node_t node;
{
  Status nodes[MAX_NODES]
  get_task_info(tracer, nodes);
  next_pro = next_priority(comp_vel);
  /* set min_con, max_con as above,
   * max_node is the node whose phase
   * is max_con */
  if (next_pro - max_con < LOW_MARK) then
    /* tracer is catching up on comp_vel:
     * schedule a comp_vel (producer) */
    schedule_task(comp_vel, max_node);
  else
    if (max_con - min_con> DIFF_TIME) then
       construct_op_distr(hint, nodes);
       /* determine the optimal distribution
        * of the consumer task */
       for i=0 to max_node do
          send_migration_hint(i, &hint[i]);
       endfor
    endif /* else: do nothing */
  endif
}
```

Figure 5: Load balancing code of LA application

the element of the task structure through task name, task pointer and field name. **ship_out_tracer** is an example of how to select what tasks should be migrated.

The load balancer uses time sharing: all nodes can execute tasks of both types. Load balancing is achieved by scheduling the next priority **comp_vel** task to the node with the largest phase value, i.e. the node that has made most progress in evaluating **tracer** tasks. If data is being consumed too fast, i.e. the difference between the priority and the phase values becomes small, we have to accelerate the data generation by scheduling more **comp_vel** task. Similarly, if the **comp_vel** task is getting ahead, we delay the scheduling of **comp_vel** tasks until the tracer tasks have caught up. The operation of the load balancer is outlined in Figure 5.

#nodes	seq.	1	2	3	4
nlb(sec)	60.5	61.53	31.94	22.19	18.16
lb(sec)	-	-	33.01	22.87	18.29

(a) homogeneous environment

#nodes	2	3	4
measured time (sec)	50.67	30.35	22.53
optimal time (sec)	47.85	26.92	18.73

(b) heterogeneous environment

Table 4: LosAngeles Air Pollution

Table 4 shows the performance results in a homogeneous and heterogeneous environment. When executed without load balancing, tracer tasks are uniformly distributed across the nodes, while the wind velocity tasks are scheduled in round robin fashion. When load balancing is used, HIGH_MARK and LOW_MARK are set to N and the DIFF_VALUE is set to N+1, where N represents the number of compute nodes. Table (a) shows the execution time in a homogeneous environment. Comparing the results with and without load balancing shows that load balancing adds little overhead. The results in a heterogeneous environment (Table (b)) include measured and estimated optimal execution times (the sequential program takes 215.32 seconds on SUN4/110). The measurements are very encouraging, certainly considering that the estimated optimal times assume no communication overhead.

6 Conclusion

We described a new load balancing architecture that applies to a wide range of applications. Our architecture simplifies the task of developing load balancers that take into account data dependencies and data locality, even for applications with heterogeneous tasks. This is achieved by providing users with a framework that allows them to develop load balancers for complex heterogeneous tasks on top of simple load balancers that are effective for homogeneous tasks with different types of data dependencies. Our performance results for both applications with homogeneous and heterogeneous tasks are very encouraging: the load balancers redistribute work while introducing relatively little overhead.

Further work is needed in several areas. First, the proposed architecture has to be evaluated using more complex applications and larger systems. Second, the interface of the current system is at a low level and the user has to specify a relatively large number of functions. Further automation is needed, and it is desirable to design a higher level interface that uses declarative definitions to define the interactions between the load balancing policies used for the different types of tasks.

References

[1] Emmanuel Arnould, Francois Bitz, Eric Cooper, H.T. Kung, Robert Sansom and Peter Steenkiste. "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM, Boston, April, 1989, pp. 205-216.

[2] Bernd Bruegge, Hiroshi Nishikawa and Peter Steenkiste, "Computing over Networks: An Illustrated Example", *Proceedings of the 6th Distributed Memory Computing Conference*, IEEE, April, 1991, pp.254-257.

[3] Thomas L. Casavant and Jon G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems", *IEEE Transaction of Software Engineering*, Vol. 14, No. 2, February, 1988, pp. 141-154.

[4] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennesy, "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors", *the 17th Annual International Symposium on Computer Architecture*, May, 1990, pp. 15-26.

[5] Reinard V. Hanxleden and L. Ridway Scott, "Load Balancing on Message Passing Architecture," *Journal of Parallel and Distributed Computing*, Vol 13, 1991, pp. 312-324.

[6] Susan F. Hummel, Edith Schonberg and Lawrence E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops", IEEE, *Supercomputing '91*, Albuquerque, November, 1991, pp. 610-619.

[7] Charles Koelbel, Piyush Mehrotra, and Jon Van Rosendale, "Supporting Shared Data Structures on Distributed Memory Architectures", *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, March, April, 1991, pp. 177-186.

[8] C. Kruskal and A, Weiss, "Allocating Independent Subtasks on Parallel Processors", *IEEE Transaction on Software Engineering*, SE-10, No. 10, October, 1985.

[9] H. T. Kung, Peter Steenkiste, Marco Gubitoso, and Manpreet Khaira, "Parallelizing a New Class of Large Applications over High-Speed Networks", *Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, March, 1990, pp. 167-177.

[10] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", *IEEE Transaction of Computers*, September, 1979, pp. 690-691.

[11] Frank C. H. Lin and Robert M. Keller, "The Gradient model load balancing method", *IEEE*

Transaction on Software Engineering, SE-13, No. 1, January, 1987, pp. 32-38.

[12] Steven Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs', *SIGPLAN '92 Conference on Programming Language Design and Implementation*, ACM, San Francisco, June, 1992, pp. 200-211.

[13] Evangelos P. Markatos and Thomas J. LeBlanc, "Load Balancing vs. Shared-Memory Multiprocessors", *21th International Conference on Parallel Processing*, August, 1992.

[14] Hiroshi Nishikawa and Peter Steenkiste, "Aroma: Language Support for Distributed Objects", In *6th International Parallel Processing Syposium*, Beverley Hills, March, 1992, pp. 686-690.

[15] Constantine D. Polychronopoulos and David J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputer", *IEEE Transaction on Computer*, C-36, No. 12, December, 1987, pp. 1425-1439.

[16] Matthew Rosing, and Robert P. Weaver. "Mapping Data to Processors in Distributed Memory Computations", *Proceedings of the Fifth Distributed Memory Computing Conference*, IEEE, April, 1990, pp. 884-893.

[17] Wei Shu and L. V. Kale, "A Dynamic Scheduling Strategy for the Chare-Kernel System", IEEE, *Supercomputing '89*, November, 1989,pp. 389-398.

[18] Ping-Sheng Tseng,"Compiling Programs for a Linear Systolic Array", *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, ACM, June, 1990, pp. 311-321.

[19] Jian Xu and Kai Hwang, "Dyanamic Load Balancing for Parallel Program Execution on a Message-Passing Multicomputer", IEEE, *2nd IEEE Symposium on Parallel and Distributed Processing*, December, 1990, pp. 402-406.