APPLICATION-LEVEL SOFTWARE SELF-BALANCING

Chung-Ta King, Tzong-Huei Shiau, Chin-Piao Chan

Institute for Integrated Systems Research
Department of Computer and Information Science
New Jersey Institute of Technology
Newark, New Jersey 07102

Abstract: In this paper we proposed a hybrid static/dynamic scheduling scheme on distributed-memory multiple-processor systems, e.g., distributed systems and multicomputers. Using this self-balancing scheme, computations are first scheduled statically, and then dynamically redistributed to adapt to the run-time environments. The rescheduling operations are directed by a number of program parameters, which can be directly accessed from within the program and will serve as processor load indices. As a result the self-balancing operations can be implemented entirely at the application level, which require minimal system supports. To illustrate the idea the self-balancing technique is applied to asynchronous iterative methods. Various design tradeoffs are discussed, and preliminary performance results on an NCUBE multicomputer are presented.

1. INTRODUCTION

In this paper we study the problem of process scheduling on distributed-memory multiple-processor systems (DMMP). A hybrid static/dynamic scheduling scheme, which can be implemented entirely at the application level, is proposed. To illustrate the idea we will apply this scheduling scheme to asynchronous iterative computations [2, 5]. However, as will be discussed later, this method can be applied to other applications as well.

The DMMP systems are those which consist of multiple loosely-coupled processors interconnected by a message-passing network and associated with distributed local memories. Distributed systems and multicomputers [1] are typical examples. DMMP systems are very suitable for resource sharing and massive parallel computations [8]. However, in such systems communication incurs non-negligible overhead. Thus, it is important to schedule the computations to the processors.

The scheduling problem is usually solved statically or dynamically. In static scheduling, computations are allocated to processors before execution starts. Although static scheduling can minimize run-time overhead, it is not always possible to gather all needed information before run-time for efficient scheduling. It is also difficult to predict and tackle dynamically changing environments, e.g., multiprogramming and communication delay. Under such situations dynamic scheduling is preferred. In dynamic scheduling computations are allocated to processors dynamically at run-time. However, dynamic scheduling may incur very high run-time overhead in maintaining the states and in operating the schedulers.

In this paper we propose a hybrid scheduling scheme called self-balancing for DMMP systems. The basic idea is to schedule the given program statically as far as possible, and then dynamically reschedule the computations to adapt to the run-time environments. To guide the rescheduling operations a number of program parameters, which are accessible directly from within the program, are chosen to serve as processor load indices.

One distinct feature of this approach is that dynamic rescheduling of the given program is accomplished entirely at the application level. Code to perform the rescheduling operations can be inserted, manually or semi-automatically, into the parallel program at or before compile time. Embedding self-balancing code into programs, however, is out of the scope of this paper.

By using application-level self-balancing, no special system supports, such as load balancers, are needed. Thus, self-balancing is immediately applicable to existing DMMP systems having no dynamic load balancing supports. Preliminary study on an NCUBE multicomputer showed that self-balancing indeed improves the performance over programs without load balancing. More experiments are now being conducted to fully understand the nature of self-balancing.

2. ASYNCHRONOUS ITERATIVE COMPUTATIONS AND THEIR EXECUTIONS

2.1. Asynchronous iterative computations

Let \( z \) be a vector of \( R^n \) and its components be denoted by \( z_i \), where \( 1 \leq i \leq n \). Let \( F \) be a function from \( R^n \) to \( R^n \). Conceptually, an iterative computation consists of a sequence of iterations, \( m = 1, 2, \ldots \). In iteration \( m \), components of \( z \) will be updated for the \( m \)-th iteration. Let \( z_i^{(m)} \) denote the value of component \( z_i \) in the \( m \)-th iteration. Then the evaluation of \( z_i^{(m)} \) at the \( m \)-th iteration is given by

\[
 z_i^{(m)} = f_i(z_1^{(m)}, \ldots, z_n^{(m)}) 
\]

where \( t_k(m) \), \( 1 \leq k \leq n \), is the iteration number in which the
value of \( z_k \) is obtained and \( f_j(x_k) \), \( 1 \leq k \leq n \), is a component function of \( F \).

An iterative computation is executed asynchronously if (a) the values used to evaluate new values are updated from time to time, and the updating, no matter how slow, should never be stopped; (b) all components of \( z \) are updated from time to time — no component's updating will be stopped forever [2]. It follows that at the same time instance different components may be in different iterations. Furthermore, there is no restriction on which values can be used in evaluating components.

Convergence conditions for general applications using iterative methods have been established in [2]. Recently, convergence conditions for nonsymmetric linear complementarity problems were given in [5].

2.2. Parallel Execution of Asynchronous Iterative Methods

A straightforward approach of executing asynchronous iterative computations on DMMP systems is to assign a fixed set of components to each processor. The problem is that this static scheduling scheme cannot handle dynamically changing run-time environments. Even for multiprocessors, on which system run-time behavior is easier to predict because processors are dedicated to one application at a time, the prediction is becoming more difficult and static scheduling is becoming less effective. New features such as multiprogramming, shared distributed-memory, concurrent I/O, and fine granularity [1, 7, 13] are introduced to meet the demands for efficient resource utilization and high performance. As a result, effective load balancing and scheduling schemes become inevitable.

Research on dynamic load balancing for DMMP systems usually concentrates on process dispatching and job migration [3, 9, 11]. These works focus on balancing the load of the whole system. Thus, they often implement load balancers or schedulers at the operating system level. Nevertheless, such mechanisms are difficult to implement, tend to be expensive, and introduce fixed scheduling overhead.

The proposed self-balancing approach considers the balance of computations within a single program. It conforms in spirit to the recent works on self-scheduling [12] for shared-memory multiprocessors. However, on DMMP systems it is expensive to implement a centralized scheduling mechanism to facilitate load-balancing. Also, processors must keep track of data allocation to ensure proper data routing and exchanging.

In the following discussion, the terms component and variable will be used interchangeably, so will be the terms processor and node.

3. SELF-BALANCING IN DMMP SYSTEMS

3.1. Basic Idea

The basic idea of the self-balancing approach is to first schedule the given program statically, and then dynamically reschedule the computations during run-time. Many static scheduling algorithms proposed previously can be adopted here [3, 4, 10]. Thus, we will concentrate on the dynamic rescheduling problem.

Consider the application of self-balancing to asynchronous iterative computations. Suppose that the asynchronous iterative method is used to solve a problem with \( n \) components on a DMMP system using \( N = (n+1) \) nodes. Also, to evaluate \( z_i \) we will need values from components \( z_{i1}, \ldots, z_{ik} \), where \( k \leq i \). Without loss of generality, assume further that, among these components, \( z_{i1}, \ldots, z_{ik} \), where \( k \leq i \), are allocated to the same processing node as \( z_i \). Initially, values of \( z_{i, k+1}, \ldots, z_u \) must be obtained from other nodes. The processor containing \( z_i \) is called a neighboring node of those processors containing \( z_{i, k+1}, \ldots, z_u \). Components \( z_{i, k+1}, \ldots, z_u \) will be called the boundary components of the processor holding them.

Given a static scheduling, the dynamic rescheduling operations in each processing node are outlined below:

1. Update all local components through \( k \geq 1 \) iterations. Throughout these \( k \) sweeps, values of \( z_{i, k+1}, \ldots, z_u \) needed for local component \( z_i \) remain the same.
2. Send local components which are needed in other nodes along with the processor's iteration number (to be discussed shortly) to the neighboring nodes.
3. Receive the most recent values of \( z_{i, k+1}, \ldots, z_u \) with their processors' iteration numbers. If a value has not arrived, use the old one.
4. Compare the received iteration numbers with the local iteration number. If the differences are above a given threshold, \( v \), then migrate \( k \) local components to or from corresponding neighboring nodes. Go to Step 1.

Steps 1-4 are repeated until all components are converged.

3.2. Design Parameters

There are several important design parameters in the above algorithm. The iteration number of a processor is chosen to serve as the load index. It can be defined as the average iteration number of all local components or as the accumulated number of iterations. In general, a faster processor will have a higher iteration number.

The parameter \( k \) defines the number of sweeps through local components before exchanging data with neighbors. It indicates the period of state information exchange. The value is chosen mainly as a tradeoff between communication and computation.

In DMMP systems, sending a message between two processors can be modeled as [6]:

\[
\sigma + s \times t
\]

where \( \sigma \) is the message startup delay, \( s \) is the length of the message, and \( t \) is the transmission delay per unit of data.
Now suppose that there are \( n \) components in a processor, and it has to transmit \( k \) components to the neighboring nodes. To process one component in one iteration would take \( q \) units of time. Then the total computation time for that \( k \)-sweep is \( nqk \). At the end of the \( k \)-sweep the processor has to send messages of a total size of \( h \) to its neighboring nodes, which takes \( m\sigma + h\tau \), where \( m \) is the number of neighbors.

A large \( k \) will limit the frequency of boundary data exchanges which slows the rate of convergence. On the other hand, a small \( k \) will reduce the ratio of computation \((nqk)\) to communication \((m\sigma + h\tau + w)\), where \( w \) is the overhead of rescheduling operations. In other words, the processor spends more time on communication and scheduling than on useful computation. From the above discussion, it seems that

\[
k = \frac{m\sigma + h\tau + w}{nq}
\]

is a good choice. Note that \( n, h, q, \) and \( w \) are problem-dependent and that \( k \) need not be fixed.

The parameter \( v \) denotes the threshold of migration. In other words, if the difference of load indices in two neighboring processors is larger than \( v \), then migration procedure is invoked. The parameter \( u \) defines the number of components transferred per migration. Both \( u \) and \( v \) will affect the rate of reaching balancing. After the migration, rescheduling operations are invoked. The parameter \( v \) defines the number of components transferred per migration. Both \( u \) and \( v \) will affect the rate of reaching balancing. Again, they need not be fixed. To gain more insight into the nature of \( k, v, \) and \( u \), a simplified model is described below.

Consider a system with two processors, \( p_1 \) and \( p_2 \). Suppose that the average time for \( p_i \) \((i=1,2)\) to process one component in one iteration, including transmission and scheduling overhead, is \( q_i \). Note that the model here is somewhat different from that in discussing \( k \). Suppose further that \( p_1 \) and \( p_2 \) have just completed one migration and \( p_i, i=1,2, \) is to start its \( k \)-th \( k \)-sweep at time \( t_i \) with \( n_i \) local components. Without loss of generality, let us assume that \( p_2 \) is faster.

The value of interest is the time when next migration will occur. Note that \( p_i, i=1,2, \) will finish \( k+1 \)-th \( k \)-sweeps at the time of \( d_i n_i q_i k + t_i \). If the condition of migration is \( d_2 - d_1 > v \), then next migration will occur at the smallest \( d_1 \) such that there exists a \( d_2 \) where

\[
(d_1 n_i q_i k + t_i) - (d_2 n_i q_i k + t_i) \leq n_2 g_2 k,
\]

and

\[
d_2 - d_1 \geq v.
\]

Note that if Eq. (2) is satisfied, then \( p_1 \) at the end of the \( k+1 \)-th \( k \)-sweeps will receive \( d_2 \) from \( p_2 \). A migration will be initiated by \( p_1 \) when \( d_2 - d_1 > v \).

After the migration, \( p_1 \) will have the following values:

\[
k' = k + d_1 + 1
\]

\[
n' = n_i - u
\]

\[
t' = d_1 n_i q_i k + t_i + w.
\]

On the other hand, \( p_2 \) will receive the migrated components and have the following parameters:

\[
k'_2 = k + 1 + \frac{t_1 - t_2}{n_2 q_2 k} + 1
\]

\[
n_2' = n_2 + u
\]

\[
t_2' = t_2 + (k'_2 - k) + n_2 q_2 k + w.
\]

These new values are then used in the next run of calculation. Under the condition that processors' speed remains stable, applying the above analysis iteratively will help to find out the rate of balancing in a two-processor system.

Table 1 shows results from the above model by applying numerical data. Let \( k_1^* \) and \( k_2^* \) denote the number of \( k \)-sweeps executed in processors \( p_1 \) and \( p_2 \), respectively, when their loads are balanced. The results are obtained assuming \( \sigma = 5, \tau = 1, n_1 = n_2 = 50, k = 2, \) and \( t = t_2 = 0 \) initially.

Table 1 confirms the intuition that a large \( v \) or a small \( v \) will increase the rate of reaching load-balancing. Adjusting \( u \) seems to be more effective. For example, comparing the three entries in Table 1, \((v,u,q_1,q_2) = (10,1.2,5.4,5),\) \((10,2.2,5.4,5),\) and \((5,1.2,5.8,5),\) we can see that doubling \( u \) results in a smaller \((k_1^*, k_2^*)\) pair at balance than dividing \( v \) to half.

### 3.3. Characteristics of Self-Balancing

Several important characteristics of self-balancing can be identified from the above discussions:

- Certain program parameters have to be selected to serve as the load index. The chosen parameters should be able to reflect the load of processing. In asynchronous iterative methods, the iteration number of the computation is used.
- No special system supports, such as load balancers, are needed. Thus, the self-balancing technique is immediately applicable to existing DMMP systems.
- Code for self-balancing can be inserted into the given program at or before compile time. As a result, load balancing can now be controlled by the programmers based on the characteristics of individual programs.
- The dynamic rescheduling procedure can complement inefficient static allocations. An initially ill-scheduled computation can be brought to balance quickly by allowing processors to redistribute their loads. Thus, the efficiency

<table>
<thead>
<tr>
<th>( v )</th>
<th>( u )</th>
<th>((q_1,q_2) = (2,5,4,5))</th>
<th>((q_1,q_2) = (2,5,8,5))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(448,364)</td>
<td>(353,363)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(190,148)</td>
<td>(210,128)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(79,55)</td>
<td>(150,93)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(56,38)</td>
<td>(83,44)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(32,20)</td>
<td>(70,37)</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>(644,500)</td>
<td>(942,641)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(351,274)</td>
<td>(370,223)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(146,102)</td>
<td>(263,161)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>(103,70)</td>
<td>(144,75)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(59,37)</td>
<td>(123,64)</td>
<td></td>
</tr>
</tbody>
</table>
of the static allocation strategy might not be a major factor.

- Self-balancing focuses on balancing the computations for individual application programs. This is different from conventional schemes, which concentrate on balancing load of the whole system.

4. SELF-BALANCING IN LINEAR ARRAYS

In this section one example is presented to illustrate implementation details and considerations of self-balancing. Consider the iterative solution to a problem in which the updating of a component depends on the values of two adjacent components. That is,

\[ y^{(m+1)}_k = f(y^{ (m)}_{k-1}, y^{ (m)}_k, y^{(m)}_{k+1}) \]

Note that this method can be applied, for example, to solve a set of linear equations where the corresponding equations embed a band matrix.

A straightforward initial allocation will employ \(N = nN \) processors configured in a linear array and assign every \( [n/N] \) consecutive components to one processor. Thus, each processor has to communicate with its two neighbors for the two boundary components.

The following symbols will be used:

- \( g(h) \) the index of the leftmost (rightmost) local component
- \( \text{Right\_diff} \) the load difference with the right (left) neighbor since last migration
- \( \text{Left\_load} \) the load of the left neighbor
- \( \text{Right\_load} \) the load of the right neighbor
- \( \text{Left\_migrate} \) the migrated load of the left neighbor
- \( \text{Right\_migrate} \) the migrated load of the right neighbor

The dynamic rescheduling program for a processor, \( p_i, \) where \( 1 \leq i \leq N, \) is as follows.

Download local variables, \( \{z_2, \ldots, z_h\}, \) together with two boundary variables, \( z_{h-1} \) and \( z_{h+1}. \)

while (not \text{Terminate}()) do

/* Main body of the iterative method */

if (not \text{Converge}()) then

\( \text{Left\_diff} = \text{Left\_diff} + k; \)
\( \text{Right\_diff} = \text{Right\_diff} + k; \)
\( \text{Send}(p_{i-1}, \text{type\_data}, \{z_2, \ldots, z_h, \text{Left\_diff}\}); \)
\( \text{Send}(p_{i+1}, \text{type\_data}, \{z_h, \text{Right\_diff}\}); \)

/* Check to see if a message from \( p_{i-1} \) has arrived */
while (\text{Check\_msg}(p_{i-1}, \text{msg\_type})) do

if (\text{msg\_type} = \text{type\_data}) then

\( \text{Receive}(p_{i-1}, \text{type\_data}, \{z_2, \ldots, \text{Left\_load}\}); \)

else if (\text{msg\_type} = \text{type\_migrate}) then

\( \text{Receive}(p_{i+1}, \text{type\_migrate}, \{z_h, \ldots, z_{h-1}\}); \)

/* \( z_{h-1}, \ldots, z_{h-1} \) are the migrated */
/* components and \( z_{h-1} \) is the new boundary */

endwhile

\( g = g - x + 1; \)
\( \text{Left\_diff} = \text{Left\_load} = 0; \)

while (\text{Check\_msg}(p_{i+1}, \text{msg\_type})) do

if (\text{msg\_type} = \text{type\_data}) then

\( \text{Receive}(p_{i+1}, \text{type\_data}, \{z_2, \ldots, \text{Right\_load}\}); \)

else if (\text{msg\_type} = \text{type\_migrate}) then

\( \text{Receive}(p_{i+1}, \text{type\_migrate}, \{z_h, \ldots, z_{h-1}\}); \)

/* \( z_{h-1}, \ldots, z_{h-1} \) are the migrated */
/* components and \( z_{h-1} \) is the new boundary */

endwhile

The program essentially follows the algorithm given in Section 3. The function \text{Converge}() returns true when all local components have converged (local convergence). The function \text{Terminate}() checks global convergence. The function \text{Check\_msg()} examines the FIFO message queue to see if a message from the specified sender has arrived. The type of the message is returned in the variable \text{msg\_type}.

To prevent the loss of migrated components, processors always receive migrated data regardless of their loads. Note that the program did not check nor correct any thrashing situation. Thrashing occurs when subsets of components are migrated back and forth among a small set of processors. A precise definition of thrashing is difficult, and more research work has to be done to fully understand the effects of thrashing.

Migration decision is made in the function \text{Decide\_migration}(). The function determines if a migration is necessary, and, if so, how many components should be migrated. Note that the values of those components which are needed in evaluating the newly migrated boundary components should also be sent. Those components now become the new boundary components.

5. SELF-BALANCING IN 2-D GRIDS

In this section we examine solution methods which arrange all components into a two-dimensional grid. In the following discussion, we will consider the simple case where
each component in the grid is updated by the following formula:
\[ z_{ij}^{(m+1)} = f_i(z_{i-1,j}^{(m+1)}, z_{i,j-1}^{(m+1)}, z_{ij}^{(m)}, z_{i+1,j}^{(m+1)}, z_{i,j+1}^{(m+1)}) \]

Initially, each processor \( p_{ij} \) in the system is allocated a block of components, e.g., \( \{z_{ij} | g1 \leq i \leq g2, h1 \leq j \leq h2\} \). As the computation progresses, components are migrated and the shapes of the components become irregular. Keeping track of local data increases the program complexity dramatically.

To reduce the program complexity, we impose the following two constraints. (1) The processors can only migrate those components which are allocated to them initially, and components which are migrated from other neighbors can only be migrated back to the original processors. (2) Migrating initially allocated data can only take place along one direction (either left-right or up-down interleaved among the processors.)

Under the above constraints, components in the processors can only take a certain shape. Fig. 1 shows one example. The dashed boxes indicate the initial allocation and solid boxes show the current allocation. The processor \( p_{ij} \) migrates data along the up-down direction and \( p_{ij+1} \) in the left-right direction. The boundary components in a processor can be easily identified by keeping four parameters: \( u_1', u_2', v_1', v_2' \) (see Fig. 1). Thus if \( p_{ij} \) in Fig. 1 is to send its boundary components to \( p_{ij+1} \), then the following program segment can be executed.

If \( v_2 = v_2' \) then
\[
\text{Send}(p_{ij+1}, \{z_{ij}|u_1 \leq j \leq u_2\})
\]
else
\[
\text{Send}(p_{ij+1}, \{z_{ij}|u_1 \leq j \leq u_2\} \cup \{z_{ij}|v_1 \leq j \leq v_2'\}) \cup \{z_{ij}|v_2 \leq j \leq v_2'\})
\]
/* See the '••' marks in Fig. 1 */

Note that in the above program segment all boundary components of a migrated region is sent back to its original processor, which is responsible for distributing those boundary components to the needing neighbors. The rest of the rescheduling algorithm is very similar to the program given in Section 4 and is omitted.

6. PERFORMANCE STUDY OF SELF-BALANCING

In this section preliminary performance results of self-balancing are presented. The program shown in Section 4 is implemented on an NCUBE multicomputer. Since processors in NCUBE are non-shared among application programs, we introduce dummy loops into the program as loads. In the experiments the load difference between each pair of adjacent processors are the same. Components are migrated in proportion to the load difference.

Preliminary results of the experiments are shown in Table 2. The values (max. load/diff.) indicates the maximum load and load difference between each pair of processors. The load is defined to be the loop count of the dummy loops. Thus, the values 300/100 in Table 2(a) means that the loads in the four processors are 300, 200, 100, and 0.

From Table 2 it can be seen that, when the system is balanced (i.e., load difference between processors is 0), self-balancing does incur only a small overhead over the one without self-balancing. However, the overall performance improvement of self-balancing is only in the range between 1% to 10%.

One explanation is that the communication overhead on the NCUBE is still too high that communication dominates our experiments. This is supported in part by the observation that the four-processor experiment (uses a larger problem size per node and thus a higher computation to communication ratio) behaves better than the eight-processor experiment.

It should be noted that self-balancing does not attempt to invent new algorithms. It just tries to improve the run-time efficiency of applications by balancing their loads. Therefore, it is unrealistic to expect dramatic performance improvement by applying self-balancing.

Table 2. Comparison of execution time between programs with and without self-balancing

<table>
<thead>
<tr>
<th>Max. load/</th>
<th>( k = 10 )</th>
<th>( k = 20 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>load diff.</td>
<td>w/o SB</td>
<td>SB</td>
</tr>
<tr>
<td>0/0</td>
<td>102090</td>
<td>104305</td>
</tr>
<tr>
<td>150/50</td>
<td>149421</td>
<td>145723</td>
</tr>
<tr>
<td>300/100</td>
<td>223199</td>
<td>211684</td>
</tr>
</tbody>
</table>

(a) 4 processors, each with 150 components

<table>
<thead>
<tr>
<th>Max. load/</th>
<th>( k = 10 )</th>
<th>( k = 20 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>load diff.</td>
<td>w/o SB</td>
<td>SB</td>
</tr>
<tr>
<td>0/0</td>
<td>75045</td>
<td>74609</td>
</tr>
<tr>
<td>120/10</td>
<td>103245</td>
<td>98589</td>
</tr>
<tr>
<td>260/30</td>
<td>180022</td>
<td>177779</td>
</tr>
<tr>
<td>350/50</td>
<td>221169</td>
<td>218032</td>
</tr>
<tr>
<td>700/100</td>
<td>401722</td>
<td>396899</td>
</tr>
</tbody>
</table>

(b) 8 processors, each with 100 components
7. CONCLUDING REMARKS

The self-balancing approach discussed in this paper has the advantage of requiring minimum system supports. The rescheduling algorithms can be implemented at the application level and incorporated into application programs at or before compile time. Using our approach, load balancing can be performed based on the characteristics of each individual application program. These features make our approach very attractive for DMMP systems.

As mentioned earlier, keeping track of data is one of the most serious challenges of self-balancing. What we need is a data structure which should not only specify component allocation and data dependencies, but also permit fast access. This will allow us to apply the self-balancing approach to problems with irregular computation patterns.

REFERENCE