Adapting workload distribution on software DSM clusters

Yen-Tso Liu¹, Tyng-Yeu Liang²,∗,† and Ce-Kuen Shieh¹

¹Department of Electrical Engineering, National Chung Kung University, No. 1, Ta-Hsueh Road, Tainan 701, Taiwan, R.O.C.
²Department of Electrical Engineering, National Kaohsiung University of Applied Sciences, No. 415, Chien Kung Road, Kaohsiung 807, Taiwan, R.O.C.

SUMMARY

Achieving an appropriate workload distribution is essential if user applications are to achieve a satisfactory performance on software distributed shared memory (SDSM) clusters. To address this problem, the present study develops a novel method for distributing program threads onto the individual computers of SDSM clusters. In contrast to alternative methods, the proposed approach takes account not only of processor speed and data-sharing aspects, but also memory availability, such that application performance can be enhanced through the implementation of appropriate workload distributions. In addition, when distributing the program threads, the proposed method specifically chooses only those computers that can enhance the performance of the application, rather than simply distributing the threads to all the available nodes in the cluster. This location policy makes it possible to specify the appropriate node combinations that optimize program performance while simultaneously maximizing resource utilization. The proposed method is implemented on a testbed referred to as Teamster. The experimental results demonstrate that, compared to alternative methods, the proposed approach delivers a 20–30% improvement in the performance of the chosen test applications. Importantly, it is shown that the proposed method can efficiently specify proper node combinations for the applications. Copyright © 2006 John Wiley & Sons, Ltd.

Received 19 April 2004; Revised 27 March 2006; Accepted 7 April 2006

KEY WORDS: distributed shared memory; workload distribution; memory availability; location policy; node combination

1. INTRODUCTION

Software distributed shared memory (SDSM) clusters utilize a set of networked computers that act as a shared-memory multiprocessor machine. Such a network-based computing environment
can benefit applications in many ways, including higher performance, higher scalability and higher adaptability. Importantly, this type of network configuration is more affordable than supercomputers or mainframe systems. Furthermore, the use of SDSM clusters dramatically reduces the complexity of programming on distributed systems. With the runtime support of SDSM systems [1–5], programmers can develop their applications on computer clusters by using shared variables rather than message passing. Consequently, they can concentrate their efforts on developing the program algorithms without the need to consider data communication issues between the various processors. As a result, SDSM clusters now provide cost-effective solutions for large-scale computational and data-intensive problems in various fields of science, engineering and commerce.

However, users still face certain problems when attempting fully to exploit the resources of an SDSM cluster. One such problem involves determining the optimal allocation of program threads to the SDSM cluster for execution purposes. The computers within SDSM clusters may well be heterogeneous in terms of their resource capabilities, computational speed, memory availability, disk space, etc. For such computing clusters, the services of an appropriate workload distribution algorithm are required to allocate program threads onto the individual execution nodes in accordance with their respective resource capabilities. In other words, nodes with more resources should be assigned a greater number of program threads. Furthermore, the resources of a computer network tend to be shared by different user applications. Consequently, the resources allocated to an individual application may vary during the course of its execution as a result of resource competition. To optimize program performance, it is necessary dynamically to adapt the workload distribution of the applications in accordance with changes in the overall network resource allocations. As the majority of users are reasonably unaware of the resource distribution status within a system and of the exact resource demands of their threads, it is clearly impractical to expect them manually to distribute or adapt the workload of their applications onto SDSM clusters. Consequently, modern SDSM systems [6–9] support a mechanism in which the workload distributions of user applications can be dynamically adapted.

Traditionally, SDSM methods [10–13] have considered only CPU power and data-sharing aspects when performing workload distribution, and have ignored the memory availability of the individual processors. However, in addition to computation and communication overheads, the cost associated with memory access is an important factor in determining the overall program performance. This issue assumes particular significance when the memory availability of a processor is insufficient to satisfy the memory demands of the local program threads. When processors are unable to supply sufficient physical memory space to cache all of the data required for the execution of their threads, they are obliged to perform page replacements for data-caching purposes, as the threads seek to access data that are not located in the local main memory. Although virtual memory technology enables the processors to complete the tasks of their local threads, the swapping latency of the associated memory accesses postpones their execution. The memory access overhead is an important factor in determining the program performance as the cost of disk accesses caused by page replacements is very expensive compared to that of data computation. Therefore, workload distribution methods that ignore the memory availability of the processors are likely to make inappropriate allocation decisions, which can actually degrade rather than enhance program performance.

Conventional program distribution mechanisms generally assign program threads to all of the nodes available in the SDSM cluster when users submit their programs for execution. However, this location policy fails both to optimize the program performance and to ensure a full utilization of the network resources. Consequently, the program execution time is unlikely to be fully minimized and the number
of execution nodes may well increase owing to the overheads of program parallelism, communication costs and memory swapping activities. If the execution time of a SDSM program increases rather than decreases as the number of cluster nodes increases, this location policy results in a waste of network resources and reduces the job throughput of the SDSM cluster. To overcome this problem, an improved solution is to select only the nodes which provide a performance enhancement when executing the application. Using this improved location policy, the SDSM cluster can significantly improve the performance of user programs with the high utilization of system resources. In this way, the remaining resources can be retained for the execution of other programs, thereby increasing the job throughput of the computer cluster.

As described above, previous methods proposed for workload distribution on SDSM clusters obviously fail to deliver an optimal program performance or to fully utilize the available resources. Consequently, the present study develops a novel workload distribution method for SDSM clusters, in which memory resource considerations are taken into account when determining the most appropriate workload distribution of a user application. This study first analyzes the impact of memory availability on program performance and then derives a set of formulae that take account of computation, communication and memory-swapping costs in predicting the execution time of a SDSM user program. Using these performance prediction formulae, an innovative algorithm, multi-combinations and multi-phases (MCMP), is constructed for mapping the working threads of user programs onto the execution nodes of SDSM clusters. This algorithm has the ability to perform a fast and effective search for proper node combination that simultaneously improves the program performance and the system resource utilization. The proposed method is then implemented on a SDSM run time system known as Teamster and is applied to two experimental applications in order to evaluate its effectiveness. The experimental results indicate that the proposed method delivers a 20–30% performance improvement compared with traditional workload distribution methods. Furthermore, the results confirm that the developed algorithm has the ability to efficiently select proper node combinations for both of the test applications.

The remainder of this paper is organized as follows. Section 2 describes previous studies related to workload distribution on SDSM clusters. Section 3 presents the proposed workload distribution method. Section 4 describes the implementation of the proposed method on the Teamster test-bed. Section 5 discusses the experimental results obtained when evaluating the effectiveness of the proposed method using two experimental applications. Finally, Section 6 presents the conclusions of the present study and highlights the direction of future research activities.

2. RELATED WORKS

Current SDSM systems supporting dynamic workload distribution include CVM, JIAJIA and Cohesion. CVM [14] focuses on obtaining a load balance and reducing data consistency communications. It achieves this goal by distributing program threads onto processors in line with the computational power of individual processors and the computational demands of the threads. Basically, processors with greater computational power are assigned a greater number of program threads. Furthermore, CVM locates pairs of threads demonstrating the highest degree of mutual data sharing on the same node in order to minimize internode communication. JIAJIA [15] assumes that the processors have sufficient physical memory space to hold the data required by their local threads, and therefore only considers CPU resource issues when determining the program workload distribution.
Adopting a similar approach to that taken by the CVM system, JIAJIA distributes the program workload in accordance with the computational power of the individual processors. The third system, Cohesion [16], divides the task of distributing the program workload into two phases, namely the migration phase and the exchange phase. In the migration phase, Cohesion estimates the appropriate workload for each processor in the same fashion as employed in the CVM system. It then migrates threads from the heavily loaded nodes to the more lightly loaded nodes in order to reduce the load imbalance cost. Meanwhile, in the exchange phase, the pairs of threads with the highest degree of mutual data sharing are located on the same node in order to reduce the communication costs induced by thread exchange.

A review of the available literature reveals that several previous researchers have investigated workload distribution issues. For example, Peris [17] analyzed the influence of physical memory size on system performance, and proposed a stochastic model that took into account memory-access costs when predicting the total execution cost of a parallel program. His study confirmed that an inappropriate workload distribution leads to increased memory access overheads, which degrade the system performance. However, although useful as a theoretical analysis tool, the model proposed in this study was difficult to realize in a practical implementation. Xiao [18] proposed the use of CPU/memory-based load sharing policies as a means of determining an optimal workload distribution. It was suggested that when a new user job is created on a machine, the distribution system should first consider the memory space available on that machine. If sufficient memory space can be provided, a CPU-based policy is then used to determine where the job should be executed, i.e. the job is executed locally if the total number of local jobs is less than the maximum number of jobs which the machine is willing to take; otherwise, the job is assigned to the remote node possessing the smallest number of jobs. Conversely, if the local node cannot provide sufficient memory space, a memory-based policy is used to locate the job, i.e. it is either located onto the node that has sufficient memory space and the least workload, or it is pended to a waiting queue. Although this method does consider the available memory resource when determining the workload distribution, its principal aim is simply to promote the full utilization of the system resources and to improve system throughput, rather than to improve the user application performance, which is the goal of the workload distribution mechanism for SDSM systems. In addition, jobs running on distributed systems tend to be independent of each other, whereas working threads running on SDSM systems must generally cooperate closely with one another when executing the same user programs. Therefore, it is clear that the method proposed by Xiao is not entirely suitable for SDSM systems.

In the preceding discussions, it has been shown that previous SDSM methods only considered the computational time and communication time of programs when distributing threads onto individual processors. In other words, they ignored the latency of the memory swapping when processors have insufficient physical memory space to cache all the data required by their local threads. Furthermore, these methods always mapped the working threads of user programs onto all of the available processors of the SDSM cluster. No account was taken of whether the performance of the user programs could be enhanced by increasing the number of execution nodes. Therefore, these methods are not particularly effective in optimizing the program performance or the resource utilization of SDSM clusters. Furthermore, the discussions above have noted that the methods proposed for distributed systems tend to be unsuitable for SDSM systems. Consequently, there exists an outstanding requirement to develop an effective workload distribution mechanism for SDSM systems.
3. THE PROPOSED METHOD

Applications running on SDSM clusters can be broadly classified into three principal categories, namely run-to-complete, iterative and fork/join [19]. A run-to-complete SDSM application executes once and then terminates. Iterative SDSM applications execute repeatedly, with barrier synchronization occurring after each execution of all the threads. Fork/join SDSM applications recursively fork new threads and wait for them to complete. As each category has different execution characteristics, it is necessary to design specific workload distribution methods for each in order to maximize program performance effectively. The current investigation chooses to address the problem of workload distribution for iterative applications, as applications of this type exhibit regular program behavior and long lives. Consequently, compared to fork/join and run-to-complete applications, the task of formulating a precise prediction of the execution time for iterative applications is more straightforward. Furthermore, there is greater merit in dynamically adapting the workload distribution of applications of this type.

3.1. Problem specification

Execution of an iterative application involves two fundamental cycles, namely execution and synchronization. In the execution cycle, the processors of the SDSM cluster work in parallel to execute the program threads created by the application. During the execution of these threads, the latency of memory swapping can be neglected if the processors have sufficient physical memory space to cache all of the data required by their local threads. However, if this condition is not satisfied, the processors are obliged to perform page replacements for data caching purposes when the threads need to access data located in the memory of a remote processor. Therefore, it is clear that the latency of memory swapping simply cannot be neglected. Once the processors finish the task of the threads in one iteration, they enter the synchronization cycle, whose purpose is to maintain the data consistency. During this cycle, the processors retrieve their updates to the shared data pages, i.e. the diffs, and then propagate these diffs to all other remote processors having the copies of the shared pages. The processors must then wait for the acknowledgement messages returned from these remote nodes to confirm that the diffs have been successfully merged into their data pages. Having completed the maintenance of data consistency, the processors send barrier-arrival messages to their root node. Once the root node has received barrier-arrival messages from all of the working processors, it then returns the resumption messages to each processor for instructing the execution of local threads in the successive iteration. Hence, the processors re-enter the execution cycle, as described above. This iteration process continues until the final iteration of the application has been successfully completed.

3.2. Process of workload distribution

The proposed method divides the process of workload distribution into an execution step and a synchronization step, in accordance with the characteristics of iterative applications described above. When an iterative application is submitted to an SDSM cluster, the application threads are usually distributed evenly onto all of the available computers of the SDSM cluster for parallel execution. After initializing the program execution, the workload distribution process enters the execution step, during which the cluster nodes execute their local threads and collect the information necessary to
adapt the workload distribution. Once all of the nodes have finished the tasks of their local threads in the first iteration, the workload distribution process enters the synchronization step. During this step, all of the execution nodes transmit their local information to the root node, i.e. the node to which the application was originally submitted. The root node then evaluates the received information and determines whether or not a workload imbalance exists. If the workloads of the individual execution nodes are balanced, the root node notifies the execution nodes to resume processing of their local threads in the next iteration, and then the workload distribution process re-enters the execution step. However, if a load imbalance exists, the root node applies an innovative thread-mapping algorithm, referred to as the MCMP algorithm, to identify a proper node combination for the available nodes of the SDSM cluster and assigns an appropriate thread-mapping pattern to this node combination for the next program execution cycle. The cluster nodes perform thread migration tasks to modify the allocation of threads in accordance with the revised thread-to-node mapping pattern generated by the MCMP algorithm. Having modified the thread allocations, the workload distribution process re-enters the execution step. Importantly, only those nodes within the new node combination which contribute to an enhanced program performance continue to work in the next iteration of the application. The remaining nodes are then released for the execution of other applications. The process described above continues iteratively until the last iteration of the application has been successfully completed.

3.3. Performance prediction formulae

The effectiveness of a thread-mapping algorithm is primarily dependent on its ability to recognize whether or not a modified thread-to-node mapping pattern is better or worse than the previous one in terms of its effect in optimizing program performance. Therefore, it is necessary to construct a set of formulae for precisely predicting the performance of an iterative application executed under a given thread-to-node mapping pattern. In contrast to previous studies, the current method incorporates the consideration of memory swapping latency costs into these performance-predicting formulae, and considers the number of processors per node when estimating the computation costs of user applications. Additionally, the proposed method adopts a ‘greedy strategy’ which aims to optimize the workload distribution for each iteration such that the total execution time of the iterative application is effectively reduced. This strategy is deliberately adopted since the amount of resource assigned to a user application may vary during the course of its execution, with the result that it is very difficult to generate precise predictions of its execution time. Therefore, the process described below for constructing the performance-predicting formulae is concerned with determining the finish time of a given iteration. Note that the symbols used in the prediction formulae and the descriptions of these symbols are all listed in Table I.

Theoretically, the finish time of a given iteration is determined by the longest finish time of any node working within that iteration, i.e.

\[ T_I = \text{Max}\{T^1, T^2, T^3, \ldots, T^N\} \]  \hspace{1cm} (1)

As described previously, the activities of a working node are divided between an execution cycle and a synchronization cycle. Therefore, the finish time of any node, \( x \), working in a given iteration, i.e. \( T^x \), comprises two basic components, namely the execution-cycle time and the synchronization-cycle time, i.e.

\[ T^x = T^x_{\text{exec}} + T^x_{\text{syn}} \]  \hspace{1cm} (2)
Table I. The description of performance symbols within the developed formulae.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N$</td>
<td>The number of execution nodes used to execute a given user application</td>
</tr>
<tr>
<td>$M$</td>
<td>The number of data pages created by a given application</td>
</tr>
<tr>
<td>$N_x$</td>
<td>The number of processors of node $x$</td>
</tr>
<tr>
<td>$M_x$</td>
<td>The number of physical memory pages of node $x$</td>
</tr>
<tr>
<td>$S_x$</td>
<td>The set of threads assigned to node $x$ for execution</td>
</tr>
<tr>
<td>$\phi(diff_{xk})$</td>
<td>The diff size of node $x$ to page $k$</td>
</tr>
<tr>
<td>$c_x(&gt;1)$</td>
<td>The computational power factor of processors of node $x$ which is relative to the reference node, i.e. a node with the lowest computational power factor, $c_{ref} = 1$</td>
</tr>
<tr>
<td>$s_x$</td>
<td>The sharing vector of node $x$, i.e. $[s_{x1}, s_{x2}, \ldots, s_{xk}</td>
</tr>
<tr>
<td>$s_{xy}$</td>
<td>The sharing vector of node $x$ and node $y$, i.e. $[s_{xy1}, s_{xy2}, \ldots, s_{xyk}</td>
</tr>
<tr>
<td>$u_x$</td>
<td>The update vector of node $x$, i.e. $[u_{x1}, u_{x2}, \ldots, u_{xk}</td>
</tr>
<tr>
<td>$t_i$</td>
<td>The amount of computational demand of thread $i$ which is relative to the reference node</td>
</tr>
<tr>
<td>$w_i$</td>
<td>The working set of the pages accessed by thread $i$</td>
</tr>
<tr>
<td>$r_{si}$</td>
<td>The average time spent by node $x$ in swapping-in a missing page</td>
</tr>
<tr>
<td>$r_{so}$</td>
<td>The average time spent by node $x$ in swapping-out a least-recently used (LRU) page</td>
</tr>
<tr>
<td>$r_{cd}$</td>
<td>The average time spent by node $x$ in creating the diff for a modified page</td>
</tr>
<tr>
<td>$r_{packet}$</td>
<td>The average time of delivering a message packet to a remote node</td>
</tr>
<tr>
<td>$r_{md}$</td>
<td>The average time spent by node $x$ in merging a diff into local memory</td>
</tr>
<tr>
<td>$W_x$</td>
<td>The working set of the pages accessed by node $x$.</td>
</tr>
<tr>
<td>$TI$</td>
<td>The finish time of a given iteration</td>
</tr>
<tr>
<td>$T^x$</td>
<td>The finish time of a node $x$ working with a given iteration</td>
</tr>
<tr>
<td>$T^x_{exec}$</td>
<td>The time spent by node $x$ at the execution cycle of a given iteration</td>
</tr>
<tr>
<td>$T^x_{syn}$</td>
<td>The time spent by node $x$ at the synchronization cycle of a given iteration</td>
</tr>
<tr>
<td>$T^x_{comp}$</td>
<td>The time spent by node $x$ in data computation</td>
</tr>
<tr>
<td>$T^x_{mem}$</td>
<td>The time spent by node $x$ in page replacements</td>
</tr>
<tr>
<td>$T^x_{cd}$</td>
<td>The time spent by node $x$ in creating diffs for shared the data pages updated by local threads</td>
</tr>
<tr>
<td>$T^x_{dd}$</td>
<td>The time spent by node $x$ in delivering diffs to other nodes for maintaining data consistency</td>
</tr>
<tr>
<td>$T^x_{ack}$</td>
<td>The time spent by node $x$ in waiting for receiving the acknowledgements of merging diffs from other nodes</td>
</tr>
</tbody>
</table>

In the execution cycle, the main task of the working nodes involves executing the local threads. This task involves data computation and data access activities. Consequently, the execution-cycle time of node $x$ can be expressed as:

$$T^x_{exec} = T^x_{comp} + T^x_{mem}$$  \hspace{1cm} (3)

Since the computational speeds of the processors may not be uniform, an estimation of the time spent on data computation for the local threads must necessarily take account of the computational power of
the respective processor. Consequently, the computation time of node $x$, i.e. $T_{\text{comp}}^x$ is given by:

$$T_{\text{comp}}^x = \frac{1}{N_x} \sum_{i \in S_x} t_i c_x$$  \hspace{1cm} (4)

Meanwhile, the memory-swapping time associated with node $x$, i.e. $T_{\text{mem}}^x$, is dependent on the memory availability at that node and the total memory demands of the local threads. If the former exceeds the latter, the latency of memory accesses can be neglected. However, if the thread requirements exceed the available memory resources, the latency of memory swapping must be incorporated into the total execution-cycle time. The latency of memory swapping is dependent on the number of page replacements required and the time involved in executing each one. The number of page replacements can be estimated from the difference between the amount of available memory resources and the amount of memory demands. Meanwhile, the time required to execute a page replacement can be divided into a swapping-out time and a swapping-in time. The swapping-out time is spent in scanning local memory pages to establish the least-recently used (LRU) page and then transferring this page from physical memory to the swap device. The swapping-in time is spent in loading a missing data page from the swap device to physical memory. Therefore, $T_{\text{mem}}^x$ can be estimated as follows:

$$T_{\text{mem}}^x = \begin{cases} 0, & \text{if } M_x \geq \phi(W_x) \\ (t_{\text{si}}^x + t_{\text{so}}^x)(\phi(W_x) - M_x), & \text{if } M_x < \phi(W_x) \end{cases}$$  \hspace{1cm} (5)

where $\phi(W_x)$ is the memory demand size of the working set of node $x$ and $W_x$ is the union of the pages accessed by the local threads executed on node $x$, i.e. $\bigcup_{i \in S_x} w_i$.

In the synchronization cycle, the main task of the cluster nodes is to maintain the data consistency. The total synchronization-cycle time of any node $x$ is determined by the degree of data sharing between this node and the other nodes. For example, if node $x$ does not share page $k$ with node $y$, node $x$ need not maintain data consistency for page $k$ between itself and node $y$. However, if this condition does not hold, node $x$ must create a diff for page $k$ and then propagate this diff to node $y$ whenever node $x$ modifies page $k$. After sending the diff, node $x$ must then wait to receive an acknowledgement message from node $y$ to confirm that the update has been successfully merged into its local memory. The waiting time mainly comprises the time spent by node $y$ in merging the diff into its local memory. Note that node $x$ creates diffs for shared pages only once even if it shares the pages with many other nodes. Therefore, the synchronization-cycle time of node $x$ can be calculated as:

$$T_{\text{syn}}^x = T_{\text{cd}}^x + T_{\text{dd}}^x + T_{\text{ack}}^x$$

$$= \sum_{k=1}^{M} s_{sk} u_{sk} t_{\text{cd}}^x + \sum_{y=1, y \neq x}^{N} \sum_{k=1}^{M} \frac{\phi(diff_{sk})}{\text{packetsize}} t_{\text{packet}} s_{xy} u_{sk} + \sum_{y=1, y \neq x}^{N} \sum_{k=1}^{M} t_{\text{md}}^y s_{xy} u_{sk}$$  \hspace{1cm} (6)

### 3.4. Thread-mapping algorithm

The problem of determining an optimal thread-to-node-mapping pattern for workload distribution to maximize the SDSM application performance is declared to be NP-complete. When the problem involves the allocation of $m$ threads to $n$ nodes, a total of $n^m$ thread-mapping patterns are possible.
Clearly, if a thread-mapping algorithm must evaluate all of these thread-mapping patterns in order to predict the optimal mapping pattern, the prediction costs will be so high as to make this algorithm impractical. Therefore, this study proposes an innovative thread-mapping algorithm, MCMP, to simplify this evaluation task. This algorithm divides the thread-mapping operation into four phases. Assume that \( n \) nodes are available. In the first phase, the algorithm creates \( n \) different-sized candidate node combinations; while in the second phase, the algorithm determines the number of threads to be assigned to each of the processors under each candidate node combination. In the third phase, the sets of threads to be assigned to each processor are determined for each candidate node combination in accordance with the results of the preceding phase. The outcome of this third phase is a set of \( n \) candidate thread-mapping patterns. In the fourth phase, the algorithm evaluates these candidate thread-mapping patterns using the performance prediction formulae described above and then selects the thread-mapping pattern that yields the minimal predicted program execution time. Consequently, the fourth step provides both a proper node combination and a suitable thread-mapping pattern for that node combination. The first three phases of the MCMP algorithm are described in detail in the following sections.

3.4.1. The first phase

The primary purpose of this phase is to qualify the possible node combinations. If a total of \( n \) available nodes exist, the number of possible node combinations is \( 2^n - 1 \). Clearly then, the cost of the thread-mapping evaluation operation increases significantly as the number of available nodes increases. However, it is not actually necessary to consider all of the possible node combinations when evaluating the mapping of the program threads onto the available nodes. If users are limited to the use of a single node, they invariably select the most powerful one to execute their applications in order to minimize the execution time. Although a somewhat simplistic approach, this location policy is nevertheless effective for the case of \( n \) nodes. Therefore, the proposed method first evaluates the power value of each available node with the following function.

\[
power(x) = \alpha c_x + (1 - \alpha)m_x, \quad (0 \leq \alpha \leq 1)
\]

(7)

In this function, \( c_x \) is the CPU power factor of node \( x \) as shown in Table I, and \( m_x \) is the memory power factor of node \( x \) that is estimated as the value of \( M_x \) divided by \( M_{\text{reference node}} \). The weight factor, i.e. \( \alpha \), is used to adjust the weights of \( c_x \) and \( m_x \) in the power value of node \( x \). The value of this weight factor is dependent on the characteristics of SDSM applications. In the experience of this study, the value of the weight factor is assigned as 0.9. After evaluating the power values of all available nodes, the proposed method sorts the available nodes in descending order according to their power values. Based on the previous location policy, the optimal node combination of size \( m \) is composed of the first \( m \) nodes of the sorted \( n \) nodes. Therefore, the MCMP algorithm effectively reduces \( 2^n - 1 \) possible node combinations to \( n \) differently-sized node combinations, and hence significantly reduces the thread-mapping cost.

3.4.2. The second phase

The aim of this phase is to adapt the number of threads assigned to each of the execution nodes in order to achieve load-balancing. In contrast to previous methods, the current approach considers not only
ability to establish proper node combination and to identify the corresponding optimal thread-mapping pattern. Although its performance rivals that of other full-search methods, its operation overheads are significantly lower. Therefore, the proposed method provides an efficient and effective approach for simultaneously optimizing the application performance and fully exploiting the available system resources.

Currently, the workload distribution mechanisms utilized by the majority of SDM clusters are embedded in the SDM run-time system. Hence, the workload distributions of individual user applications are processed in isolation from one another. However, if the performance of the applications is to be individually optimized, and the job throughput of the cluster enhanced, it is necessary to consider the resource demands issued by each of the applications simultaneously. Therefore, it is the current authors’ intention to design and construct a resource manager to separate the workload distribution mechanism from the SDM run-time system in a future study.

REFERENCES


