An Approach of Chunk-based Task Runtime Prediction for Self-Scheduling on Multi-core Desk Grid

Peifeng Li, Qiaoming Zhu, Qin Ji, Xiaoxu Zhu
School of Computer Science and Technology, Soochow University, Suzhou, China, 215006
{pfli, qmzhu, qinji, xxzhu @suda.edu.cn}

***Abstract*—Self-Scheduling is a dynamic and adaptive loop scheduling approach to reduce the total execution time for a task running in the cluster or grid environment. This paper focuses on how to use and optimize Self-scheduling technologies to allocate tasks reasonable and achieve better parallel performance. It introduces the prediction algorithms and proposes a novel Chunk-based Task Runtime Prediction (CTRP) algorithm according to the characters of desk grid and multi-core environment. Our experimental results show that our approach can predict the execution time more accurate and achieve better load balancing than that of others when most slave nodes' load is changing frequently and rulelessly in the multi-core desk grid.**

***Index Terms*—Self-Scheduling, Chunk-based Task Runtime Prediction, desk grid**

I. INTRODUCTION

Desk grid is mainly composed of many personal computers and it's more complex and dynamic than others. Using vacant computation abilities in desk grid can provide an inexpensive and convenient solution to solve large scale computational problems, so that how to allocate tasks to these nodes and achieve better load balancing is an issue in that field. Self-scheduling [1, 2], a dynamic, adaptive and distributed scheduling method which works on application layer, is one of accepted solutions. It divides a task into a set of subtasks, and then parallels them on those computing nodes.

Currently, the existed self-scheduling schemes can achieve good performance in homogeneous environments or cluster systems. However, the load unbalancing will occur and the total execution time will increase greatly if these schemes are applied to the desk grid [3]. The main reason of that is the load of desk grid is changing frequently and irregularly, unlike homogeneous environments or cluster systems. Unfortunately, these schemes always suppose the node is dedicated by them, so their assignment policies just depend on the hardware and the historical execution information. Therefore, regardless of the factor of load, they will cause the load unbalancing and increase the total execution time greatly.

This paper focuses on how to use and optimize self-scheduling technologies to allocate tasks reasonable and achieve better parallel performance. It introduces the prediction algorithms and proposes a novel Chunk-based Task Runtime Prediction (CTRP) algorithm according to the characters of desk grid and the multi-core systems. That algorithm can predict execution time more accurate and achieve better load balancing than that of others when most nodes' load is changing frequently and rulelessly in the multi-core desk grid.

This paper is organized as follows. Section 2 describes the Self-scheduling model and its related works. Section 3 introduces the methodology of our scheme. Section 4 gives the approach to predict the size of prediction chunk. Section 5 reports the experimental results. Section 6 draws the conclusion.

II. RELATED WORKS

Self-scheduling is a data parallel method and it divides the data to be executed into many chunks with same or different size. When the idle nodes request new chunk, the master assigns one to them. For this, the Master-Slave architecture [4, 5] used in the self-scheduling is showed as Fig. 1.

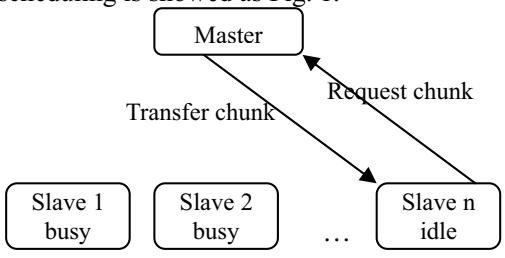


Figure 1. Architecture of self-scheduling

In Fig. 1, master node is a scheduler and it assigns chunks to slave nodes to parallel. Idle slave nodes communicate a request to the master for new chunk. The size of chunk a node should be assigned is an

doi:10.4304/jcp.6.7.1339-1345

important issue in self-scheduling. Due to nodes heterogeneity and communication overhead, assigning the wrong node a large chunk at the wrong time may cause load unbalancing. Also, assigning a small chunk may cause too much communication and scheduling overhead.

In a common self-scheduling scheme, at the *i*-th scheduling step, the master computes the chunk size C_i and the remaining number of task R_i:

$$R_0=N, \quad C_i=f(i,P), \quad R_i=R_{i-1}-C_i \qquad (1)$$

where N is the total data to be executed, $f(i,P)$ possibly has more parameters than just i and P and it is a function to produce the chunk size at each step, and P is the number of nodes on the grid. The master assigns C_i task to an idle slave and the load balancing will depend on the execution time gap between T_i, for $i=1...n$. This gap may be large if the first chunk is too large or (more often) if the last chunk (called the *critical chunk*) is too small [6].

The different ways to compute C_i has produced many self-scheduling schemes in the cluster. These schemes address how to divide a task to chunks and then dispatch them to each idle slave node in order to achieve load balancing and reduce the scheduling overhead. Some notable schemes are listed as follows.

Chunk Self-Scheduling (CSS) [7]: $C_i = k$, where $k >=1$ is the chunk size chosen by the user. For $k = 1$, CSS is the so-called Pure Self-Scheduling (PSS).

Guided Self-Scheduling (GSS) [8]: $C_i = [R_{i-1}/p]$ where p is the number of slave nodes in the grid and R_i is the remaining number of task at i-th step.

Trapezoid Self-Scheduling (TSS) [9]: $C_i = C_{i-1} - D$ with chunk decrement $D = \left\lfloor \frac{F-L}{M-1} \right\rfloor$ where the first and last chunk-sizes (F, L) are user/compiler-input or $F = \left\lceil \frac{N}{2p} \right\rceil$, L=1. p is the number of slave nodes and M is the number of scheduling steps.

Factoring Self-Scheduling (FSS) [10]: $C_i = [R_{i-1}/(\alpha p)]$ where the parameter α is computed by a probability distribution or is suboptimal chosen α =2. p is the number of slave nodes and R_i is the remaining number of task at i-th step.

To fit the characteristic of grid, some dynamic self-scheduling schemes are proposed to achieve the load balancing and reduce the scheduling overhead.

α Self-Scheduling [11]: α% partition of workload is dispatched according to their performance weighted by CPU clock in the first phase and the rest (100-α%) of workload is dispatched according to known self-scheduling in the second phase.

Hybird Parallel Loop Scheduling (HPLS) and it's optimized algorithms [12, 13, 14] : α% partition of workload is dispatched according to the performance ratio of all nodes in the first phase and the rest workload is dispatched by some known self-scheduling in the second phase. HPLS is an optimization of α self-scheduling while it considered the performance of all nodes and the bandwidth between each node and the master. This approach has two problems when applied to the grid system.

Firstly, the parameter α value is assigned before the scheduling and it's fixed during the processing. The grid is a dynamic computing environment, if the node set of the grid system is changed, the predefined α value will be out of time.

Secondly, the rest (100-α%) of workload is dispatched according to known self-scheduling in the second phase, so that it will cause the load unbalancing because those known self-scheduling (CSS, GSS, FSS, etc), didn't consider the performance of all nodes.

ACSS (Adaptive Chunk Self-Scheduling Scheme) [3]: It propose an approach to parallel the data transfer and data processing by using the multiple threads mechanism to reduce the waiting time firstly, and then it adjusts scheduling parameters dynamically based on the node's performance to achieve shorter waiting time than other schemes. In the serial processing mode the slave spends more waiting time on waiting the new data, and that would increase the total execution time and depress the efficiency of slave nodes. ACSS provides a parallel processing mode to parallel the data transfer and data processing, so that it can reduce the execution time significantly.

III. METHODOLOGY

In the self-scheduling, the master doesn't transfer the data to the slave just one time. To achieve shortest execution time in it, the master must assign a chunk with appropriate size to each slave on their processing ability every time. If all slaves finish their task on the same time and they are busy during the scheduling, the scheme can achieve the best performance. Therefore, how to get the slave's processing ability is an important issue.

In the desk grid, the slave's processing ability is dynamic and changed with its load. For the chunk with same size, the slave need more time to execute it if there are many processes running at that time; on the contrary, if there are just two or three running processes the slave only need less time to complete that chunk. So how to obtain the real-time processing ability of the slave and assign an appropriate size chunk to it is one of the key to reduce the execution time.

For a fixed size task, the real-time processing ability of the slave reflects in its execution time. Therefore, the execution time of a fixed task on a node can represent its real-time processing ability. Currently, there are many researches on predicting the task execution time in the grid environment and two methods are used to acquire it. 1) Static method [11, 12]. This method just collects all kinds of computer's hardware configuration, such as CPU, memory,

network bandwidth, etc, and then uses an equation to combine them; 2) Dynamic method. One method [3, 13, 14] is to run a program on the slave firstly, and then collects the execution information to obtain the slave's execution time. Another method [15, 16, 17, 18] is to predict the future execution time based on the previous running information and some similar patterns and its basic approaches are Mean Value Method, Exponential Average Method, KNN (K-Nearest Neighbor), etc.

The main disadvantage of above two methods is that they suppose the slave just runs one task and then miss the load change in it. Different with the other homogeneous environments or cluster systems, desk grid is built by many PCs used as desktop and the load of them are ruleless for the foreground processes. Ignoring the load change on the slaves would lead to the load unbalancing for self-scheduling.

According to the characteristic of load change on the slave, we propose a novel runtime (also called as execution time) prediction algorithm - Chunk-based Task Runtime Prediction (CTRP) which predicts the execution time on the real-time information, not the historical information. The principle of our approach is that the load change on the slave can't be estimated for its irregularity in a long period. But in a short period, such as several minutes, the load of the slave is steady relatively. Therefore, predicting the future runtime on the real-time information is more reliable than that on the historical information.

CTRP's theory is as follows. Each chunk assigned to the slave is divided into two parts: prediction chunk and remaining chunk. After the slave has finished the prediction chunk, it sends the real-time execution information to the master. Then the master predicts the size of next chunk and next prediction chunk based on that real-time execution information from the slave and master. After that, it transfers the next chunk to the slave. In the multi-core environment, multi-thread is used to parallel the data transfer and processing on the slave to ensure it doesn't need to wait the next chunk. This means the data transfer should be finished before the slave completes the remaining chunk.

Fig. 2 is the executing flow for *i*-th chunk based on multi-thread. We propose an approach to parallel the data transfer and data processing by using the multiple threads mechanism and there are three threads running on the slave: processing thread (PT), receiving thread (RT) and sending thread (ST). Firstly, PT processes the prediction chunk and obtains some real-time execution information. Secondly ST sends the results to the master. Thirdly, the master receives the results at the same time and then predicts the size of next chunk and next prediction chunk. Finally, the slave receives the next chunk from the master. In the multi-core environment, PT, ST and RT can be paralleled and that model can reduce the waiting time of the slave and keep it busy almost all time.

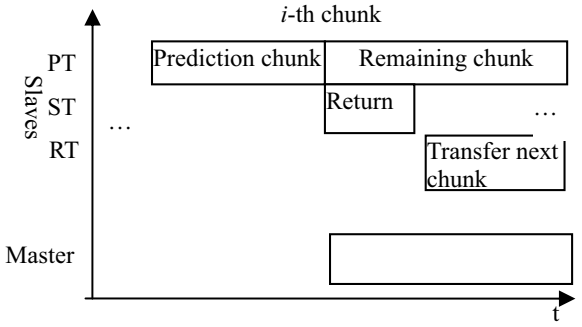


Figure 2. The executing flow for *i*-th chunk based on multi-thread

IV. PREDICTING THE SIZE OF PREDICTION CHUNK

It's obvious that how to determine the prediction chunk size is an issue for our approach. Considering the prediction chunk size by oneself, larger it is, more accurate the execution time of remaining chunk is. However, if the prediction chunk size is too large, the time to complete the remaining chunk is too small to transfer the next chunk to the execution node. Therefore, PT must wait and the execution time would be increased.

An appropriate size of the prediction chunk is to keep the node busy and let the master have enough time to transfer the next chunk to the slave node. There are four factors involved with the size of the prediction chunk, including:

- T_{rm}: The execution time of remaining chunk;
- T_{re}: The time to return the result from the slave to the master;
- T_{om}: The overhead of the master to prepare the transfer;
- T_{tr}: The time to transfer the data from the master to the slave.

To keep the slave busy, the master must have transferred the next chunk to the slave before it completed the remaining part of current chunk. Therefore, those four factors must be satisfied with the equation (2):

$$T_{rm} >= T_{re} + T_{om} + T_{tr} \quad (2)$$

The time T_{re}, T_{om} *and* T_{tr} depend on the network transfer rate, the size of the data to transfer, the load of the master and the load of the slave. We use the following method to collect the real-time information and then to calculate the T_{re}, T_{om} *and* T_{tr}.

(1) Assume half of the first chunk as the prediction chunk;

(2) Choose the total chunk size *ct* following the CHSS scheme and then assigns the data with the size *ct* to all slaves. The first chunk for slave PR_i is assigned to $ct * apf_{i,1}$ and the performance ratio $apf_{i,1}$ is defined as

$$apf_{i,1} = w_1 \times \frac{1/et_i}{\sum_{\forall node_j \in S} 1/et_j} + w_2 \times \frac{bw_i}{\sum_{\forall node_j \in S} bw_j} \quad (3)$$

where
S is the set of all grid nodes;
bw_i is the bandwidth between PR_i and the master;
et_i is the execution time of PR_i for a special program used to test the computing performance;
w_1 is the weight of the first term;
w_2 is the weight of the second term.
w_3 is the weight of the third term;

In our experiments, the parameters of w_1, w_2 and w_3 are assigned as 0.7, 0.2 and 0.1 by the linear regression on our experiments. Furthermore, et_i for PR_i is acquired by executing the matrix multiplication for size 256×256 *while* bw_i is gathered by a combination of PING and FTP.

(3) For each slave PR_i
   a) The master transfers the j-th chunk with the size $ct^* apf_{i,j}$ to it and also sends it the size of j-th prediction chunk;
   b) PR_i receives the j-th chunk from the master and then processes it. After PR_i has finished the j-th prediction chunk, it sends $Ts_{i,j}$, $Tex_{i,j}$ and $SL_{i,j}$ to the master where $Ts_{i,j}$ is the transfer time to send the j-th chunk from the master to PR_i, $Tex_{i,j}$ is the execution time PR_i used to process the j-th prediction chunk, and $SL_{i,j}$ is the average CPU utilization ratio when PR_i processes the j-th prediction chunk.

(4) If j-th chunk is the last chunk, then it's end;

(5) If the master has received the $Tr_{i,j}$, $Tex_{i,j}$ and $SL_{i,j}$ from PR_i, it also obtians $ML_{i,j}$ which is the current CPU utilization ratio in the master.

(6) The master calculates the performance ratio $apf_{i,j+1}$ as

$$apf_{i,j+1} = w_1 \times \frac{Ts_{i,j} / (ct * apf_{i,j})}{\sum_{\forall node_k \in S} Ts_{k,j} / (ct * apf_{k,j})} + w_2 \times \frac{bw_k}{\sum_{\forall node_k \in S} bw_k} \quad (4)$$

(7) The master estimates the $T_{re_{i,j+1}}$ *and* $T_{tr_{i,j+1}}$ following the equation (5) and (6):

$$T_{re_{i,j+1}} = \frac{Ts_{i,j} / (ct * afp_{i,j})}{1 - ML_{i,j}} * (ct * afp_{i,j+1}) = \frac{Ts_{i,j} * afp_{i,j+1}}{(1 - ML_{i,j}) * afp_{i,j}} \quad (5)$$

$$T_{tr_{i,j+1}} = \frac{Ts_{i,j} / ct * afp_{i,j}}{1 - SL_{i,j}} * (ct * afp_{i,j+1}) = \frac{Ts_{i,j} * afp_{i,j+1}}{(1 - SL_{i,j}) * afp_{i,j}} \quad (6)$$

where $T_{re_{i,j+1}}$ is the estimated time to return the *(j+1)*-th results from PR_i to the master while $T_{tr_{i,j+1}}$ is estimated time to transfer *(j+1)*-th chunk from the master to PR_i. $T_{om_{i,j+1}}$, the estimated overhead of the mater to prepare *(j+1)*-th chunk for PR_i, is equal to current overhead.

(8) Following the equation (2), (5) and (6), the execution time to process *(j+1)*-th remaining chunk $T_{rm_{i,j+1}}$ is

$$T_{rm_{i,j+1}} \geq \frac{Ts_{i,j} * afp_{i,j+1}}{(1 - SL_{ij}) * afp_i} + \frac{Ts_{i,j} * afp_{i,j+1}}{(1 - ML_{ij}) * afp_i} + T_{om_{i,j+1}} \quad (7)$$

and the size of *(j+1)*-th remaining chunk $Cr_{i,j+1}$ is

$$Cr_{i,j+1} = T_{rm_{i,j+1}} * \frac{Cp_{i,j}}{Tex_{i,j}} \quad (8)$$

where $Cp_{i,j}$ is the size of j-th prediction chunk.
The size of *(j+1)*-th prediction chunk $Cp_{i,j+1}$ is

$$Cp_{i,j+1} = ct^* \ apf_{i,j+1} - Cr_{i,j+1} \quad (9)$$

(9) Go to (3).

V. Experimental Results

To verify our approach, we test it on a test bed grid – ZHHZGrid [19, 20]. ZHHZGrid is a desk grid oriented to process natural language and provides some common services (parsing, name entity recognition, part of speed, word segmentation, etc) in natural language processing. That grid is built by more than 60 PCs and servers which are used by our graduate students. The test bed includes 40 multi-core PCs, one of them being assigned the role of master. We want to test our approach in the heterogeneous computing environment, so we apply a combination of computer types in three subnets and the hardware configurations of those PCs are various.

The matrix multiplication and Chinese word segmentation are used as examples while the former is a CPU-bound application and the latter is an I/O-bound application. For the CPU-bound application, the total execution time mostly depends on each node's execution time. By contrary, the total execution time mainly consists of each node's execution time and transfer time for the I/O-bound application. We test our approach on these two types application.

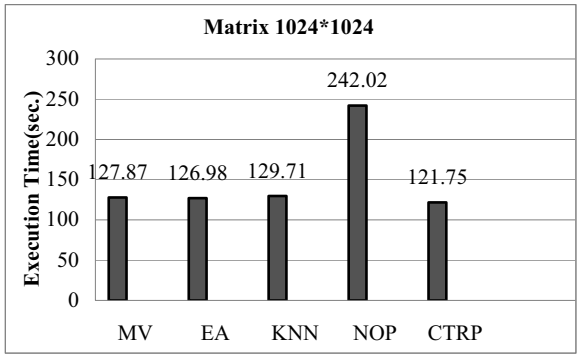
The grid middleware is Globus Toolkit 4.0 and all services are written in JAVA. We have implemented the Matrix Multiplication Service and the Word Segmentation Service and deployed them on all nodes.

We apply four runtime prediction approaches (Mean Value Method (MV), Exponential Average Method (EA), KNN and chunk-based task runtime prediction) to our self-scheduling scheme ACSS. In that experiment, k in the CSS is assigned to 12 and 100KB in matrix multiplication and Chinese word segmentation respectively.

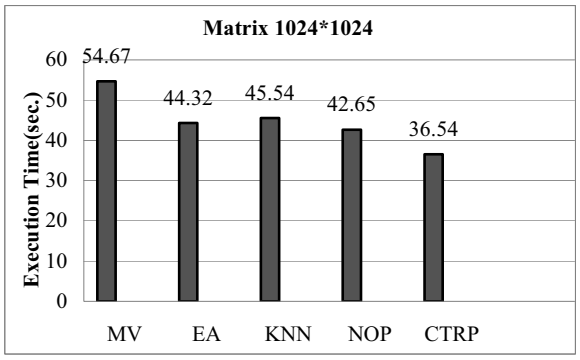
To verify all prediction approaches, we provide two

test environments: dedicated model and no-dedicated model. In the dedicated model, each node just runs our test task and the load of such node is steady. In the no-dedicated model, each node runs many tasks and the load change is frequent and irregular.

Firstly, we test above four prediction approaches and the CHSS without prediction (NOP) on the matrix multiplication with input matrix size 1024×1024 and Chinese word segmentation with text size 16MB. Fig. 3 and 4 illustrate average execution time (the average of five tests) of above five schemes. Otherwise, all 40 PCs are running all kinds of foreground threads (MS word, Internet Explorer, etc) during the no-dedicated test.

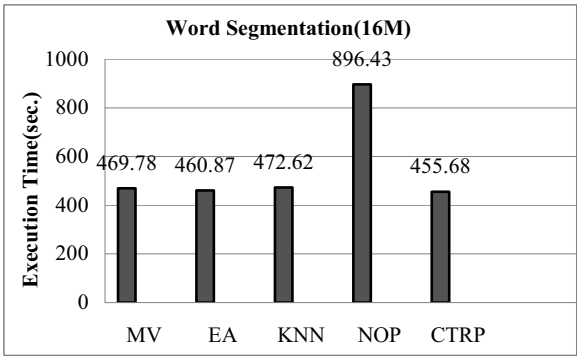


a) Dedicated model

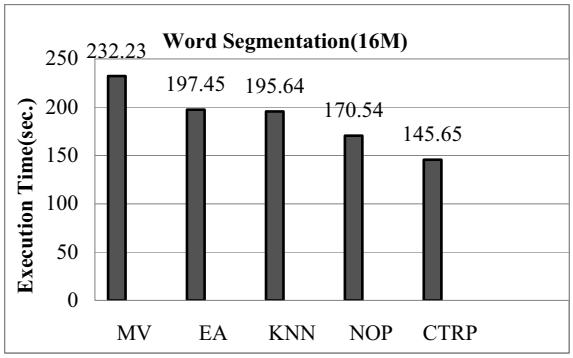


b) No-dedicated model

Figure 3. Execution time of matrix multiplication



a) Dedicated model



b) No-dedicated model

Figure 4. Execution time of Chinese word segmentation

Experimental results showed that our approach got better performance than that of others. In the case of dedicated environment, our scheme got 4.8%, 4.1%, 6.1% and 49.7% performance improvement over that of MV, EA KNN and NOP respectively in the test of matrix multiplication and also got 3.0%, 1.1%, 3.6% and 49.2% performance improvement over them respectively in the test of Chinese word segmentation. That results show that the runtime prediction approach is benefit to the self-scheduling and can reduce the execution time. In the dedicated environment, the load of each node is almost steady and the prediction approach based on historical execution information, such as exponential average, KNN, etc, also can predict the execution time correctly.

In the case of no-dedicated environment, our scheme improves the performance significantly. It got 33.2%, 17.6%, 20.0% and 14.3% performance improvement over that of MV, EA KNN and NOP in the test of matrix multiplication and also got 37.3%, 26.2%, 25.6% and 14.6% performance improvement over them respectively in the test of Chinese word segmentation. In such environment, the load of each node is changing frequently and irregularly, so that the error of all the prediction approaches based on historical execution information are increasing and their performance is worse than that of the approach with no prediction. As for our approach, it's based on current running information, so it fits the load change on every node and gets more small error.

Then, we apply our prediction approach to three classic self-scheduling (CSS, HPLS and CHSS) and call them as PCSS, PHLPS and PCHSS. We test those six scheme on the matrix multiplication with input matrix size 1024×1024 and Chinese word segmentation with text size 16MB in the no-dedicated model. Fig. 5 and 6 illustrate average execution time (the average of five tests) of above six schemes. In consideration of the fairness, we also reconstruct the CSS and HLPS with our parallel model.

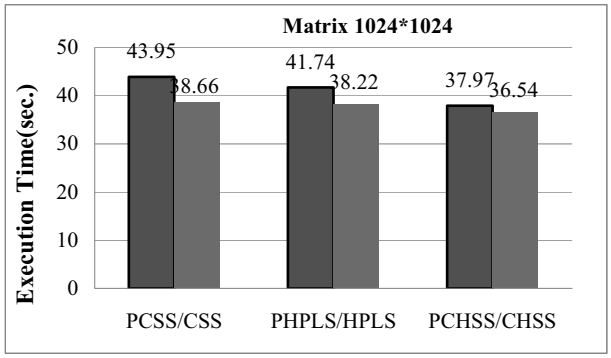


Figure 5. Execution time of matrix multiplication on six schemes

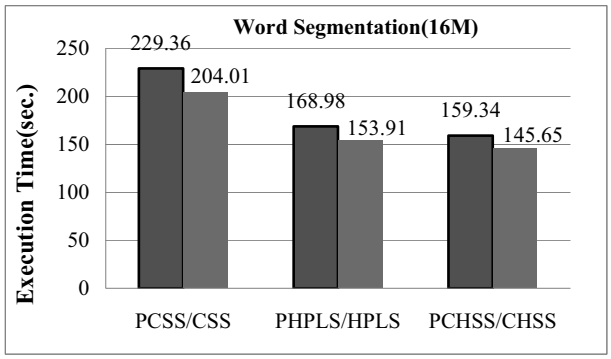


Figure 6. Execution time of Chinese word segmentation on six schemes

In Fig. 5 and 6, we found out that our prediction approach also can improve the performance of other schemes besides CHSS. Our approach CTRP got 12.0%, 8.4%, 3.8% performance improvement on CSS, HPLS and CHSS n the test of matrix multiplication and also got 11.0%, 8.9%, 8.6% performance improvement over them in the test of Chinese word segmentation. This experimental results show that our prediction approach is useful for the multi-core desk grid.

VI. Conclusion

In this paper we have proposed a novel runtime prediction approach – CTRP for the self-scheduling on the multi-core desk grid. CTRP predicts the execution time on the current real-time execution information and node's load, not the historical information. It meets the real load change on the multi-core desk grid. The experimental results showed that our approach could reduce the total execution time and achieve better load balancing than other similar prediction schemes. Our future work will focus on how to apply it to the cloud computing environment.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments on this paper. This research was supported by the National Natural Science Foundation of China under Grant No. 61070123 and No. 60873150, the Natural Science Major Fundamental Research Program of the Jiangsu Higher Education Institutions, China under Grant No. 08KJA520002.

References

[1] E. P. Markatos and T. J. Leblanc, Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessor, *IEEE Trans. on Parallel and Distributed System*, Vol. 5, No. 4, April 1994, pp. 379-400.

[2] Y. Yan, C. Jin, X. Zhang, Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems, *IEEE Trans. on Parallel and Distributed System*, Vol. 8, No. 1, Jan. 1997, pp. 70-81.

[3]P. Li, Q. Ji, Y. Zhang, Q. Zhu, An Adaptive Chunk Self-Scheduling Scheme on Service Grid, In *Proc. of the 3rd IEEE Asia-Pacific Services Computing Conference (APSCC 2008)*, Yilan, Taiwan, 2008, pp. 39-44.

[4] A. T. Chronopoulos, S. Penmatsa, N. Yu, D. Yu, Scalable Loop Self-scheduling Schemes for Heterogeneous Clusters, *Int. Journal of Computational Science and Engineering*, 2005, Vol. 1, No. 2/3/4, pp. 110 - 117.

[5] A. T. Chronopoulos, M. Benche, D. Grosu, R. Andonie, A Class of Loop Self-Scheduling for Heterogeneous Clusters, In *Proc. of the 3rd IEEE Intl. Conf. on Cluster Computing*, 2001, pp. 282-291.

[6] P. Tang and P. C. Yew, Processor Self-scheduling for Multiple-nested Parallel Loops, In *Proc. of the 1986 Intl. Conf. on Parallel Processing*, 1986, pp. 528-535.

[7] M. Benche and D. Grosu, A Class of Loop Self-Scheduling for Heterogeneous Clusters, In *Proc. of the 2001 IEEE Intl. Conf. on Cluster Computing*, 2001, pp. 282.

[8] C. D. Polychronopoulos and D. Kuck, Guided Self-Scheduling: a Practical Scheduling Scheme for Parallel Supercomputers, *IEEE Trans. on Computer*, Vol. 36, Dec. 1987, pp. 1425-1439.

[9] S. F. Hummel, E. Schonberg, L. E. Flynn, Factoring, a Method for Scheduling Parallel Loops, *Communications of the ACM*, Vol. 35, No. 8, 1992.

[10] T. H. Tzen and L. M. Ni, Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers, *IEEE Trans. On Parallel and Distributed System*, Vol. 4, No. 1, Jan. 1993, pp. 87-98.

[11] C. Yang, K. Cheng, K. Li, An Efficient Parallel Loop Self-Scheduling on Grid Environments, In *Proc. of 2004 IFIP Intl. Conf. on Network and Parallel Computing*, LNCS 3222, 2004, pp. 92-100.

[12] W. Shih, C. Yang, S. Tseng, A Hybrid Parallel Loop Scheduling Scheme on Grid Environments, In *Proc. of the 4th Intl. Conf. on Cooperative Computing*, LNCS 3795, 2005, pp. 370-381.

[13] C. Yang, K. Cheng, W. Shih, On Development of an Efficient Parallel Loop Self-Scheduling for Grid Computing Environments, *Parallel Computing*, 2007,

Vol. 33, pp. 467-487.
[14]C. Wu, L. Huang, L. Lai, M. Chen, Enhanced Parallel Loop Self-Scheduling for Heterogeneous Multi-Core Cluster Systems, In *proc. of 10th Int. Sym. on Pervasive Systems, Algorithms, and Networks*, 2009, pp. 568-573.
[15] L. Yang, I. Foster, J. M. Schopf, Homeostatic and Tendency-Based CPU Load Predictions, In *Proc. of the 17th Int. Sym. on Parallel and Distributed Processing*, April 22-26, 2003, pp. 42.2.
[16] W. Smith, I. Foster, V. Taylor, Predicting Application Run Times with Historical Information, *Journal of Parallel and Distributed Computing*, Vol. 64, No. 9, 2004, pp. 1007-1016.
[17] Y. Zhang, W. Sun, Y. Inoguchi, Predict Task Running Time in Grid Environments Based on CPU Load Prediction, *Future Generation Computer Systems*, Vol. 24, No. 6, 2008, pp. 489-497.
[18] X. Che, L. Hu, D. Guo, Information Service Prototype System for Run-time Prediction of Grid Applications . In *Proc. of the 2nd Int. Conf. on Pervasive Computing and Applications*, 2007, pp. 530-535.
[19] P. Li, Q. Zhu, L. Zhi, Design of Grid Resource Management System Oriented to Information Service, *Computer Engineering*, 2008, Vol. 34, No. 3, pp. 49-51, 58.
[20] Q. Zhu, P. Li, Z. Gong, L. Xu, ADJSA: An Adaptable Dynamic Job Scheduling Approach Based on Historical Information, In *Proc. of the 2nd Intl. Conf. on Scalable Information Systems*, 2007.

Peifeng Li, born in Jiangsu Province, China, in 1979. He has received his B.S., M.S. and Ph.D. degrees in computer science from Soochow University in 1994, 1997 and 2006 respectively. Currently he is an Associate Professor in school of computer science and technology of Soochow University since 2004. His research interests include computer network, distributed computing, cloud computing etc.

Qiaoming Zhu, received his Ph.D. degree in computer science from Soochow University in 2006. He is now a Professor of school of computer science and technology of Soochow University. His research interests include grid computing, cloud computing, natural language processing, etc.

Qin Ji, received her M.S. degrees in computer science from Soochow University in 2008. Her research interests include grid computing, cloud computing, etc.

Xiaoxu Zhu, received his M.S. degrees in computer science from Soochow University in 2004. He is now working towards his Ph.D. degree in computer science at Soochow University. His research interests include computer network, distributed computing, cloud computing, etc.