2012 IEEE 11th International Symposium on Network Computing and Applications

Two-Dimensional Dynamic Loop Scheduling Schemes for Computer Clusters

Anthony T. Chronopoulos*, Satish Penmatsa†

Naveen Jayakumar*, Eric Ogharandukun†

*Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, USA
atc@cs.utsa.edu, njayakumar5986@gmail.com

†Department of Mathematics & Computer Science
University of Maryland Eastern Shore
Princess Anne, MD, USA
{spenmatsa, eeogharandukun}@umes.edu

***Abstract*—Efficient scheduling of parallel loops in a network of computers can significantly reduce the total execution time of complex scientific applications. In this paper, we compare the performance of two-dimensional dynamic loop scheduling schemes for computer clusters with that of one-dimensional loop scheduling schemes. The loop scheduling schemes are implemented using the Message Passing Interface on a cluster of processors. Experimental results show that the two-dimensional scheduling schemes were found to significantly reduce the total execution time of tasks over the one-dimensional schemes. In addition, the two-dimensional schemes present a more balanced load distribution of the workload among the computers in the cluster.**

***Keywords*-Dynamic scheduling; cluster computing; parallel loops.**

I. Introduction

A computer cluster consists of a set of computers (resources or processors) connected to each other through fast local area networks commonly used for executing computation-intensive scientific applications. In a distributed processing cluster, the tasks are broken down and allocated to the available computers for parallel execution in order to minimize the total execution time of an application. Appropriate software runs within this network of computing resources to allow for inter-process communication between the various resources by use of Message Passing Interface (MPI).

Scientific applications usually consist of large loops (repeated execution of a set of statements) inside them. These loops are one of the largest sources of parallelism. Instead of executing an application with large loops on a single computer, if the loop iterations are divided and allocated to the available computers (processors) in the cluster, the total execution time of the application can be significantly reduced. In the ideal case, the execution time can be reduced by a factor of the number of available processors in the cluster. Hence, efficient loop scheduling schemes are essential for executing parallel applications on computer clusters.

If the iterations of a loop have no interdependencies, each iteration can be considered as a task and can be scheduled independently. Such parallel loops are often called DOALL loops. The loops that have interdependencies are often called DOACROSS loops. Several loop scheduling schemes for DOALL and DOACROSS loops have been studied in the past. For example, please see [1][2][3] and references there-in.

The loop scheduling schemes suitable for homogeneous systems are called 'simple' schemes whereas the schemes that take the heterogeneity of the system into account are called 'distributed' schemes and are suitable for heterogeneous systems. Also, loop scheduling can be categorized into 'static' and 'dynamic'. Static scheduling schemes determine the task allocation to the processors prior to the execution of the application. Dynamic scheduling (or self-scheduling) is an automatic loop scheduling method in which idle processors request new loop iterations to be assigned to them during run time (the execution of the application).

Some notable DOALL loop scheduling schemes for homogeneous and heterogeneous systems have been studied in [2][3][4]. DOACROSS loop scheduling schemes for homogeneous and heterogeneous systems have been studied in [5] and references there-in. Recent research results have been reported for designing loop self-scheduling methods for multi-core, graphics processing unit, grid, and cloud systems. Please see [6][7][8][9][10] [11][12][13].

Most of the previously studied loop scheduling schemes partition only the outermost loop of a program loop structure and assign tasks (chunks of iterations) to the processors. This is not efficient for multi-dimensional nested loops. All the previous multi-dimensional loop scheduling schemes for nested loops are static. Thus, these methods are inefficient when the loop tasks sizes are unequal. Results on two-dimensional (2-D) Trapezoid loop scheduling were presented in [14].

Here, we implement some well-known one-dimensional (1-D) loop scheduling schemes in 2-D form. We then compare the performance of the 2-D versus the 1-D schemes. We implemented the distributed loop scheduling schemes (1-D and 2-D versions) (with the Master-Worker architecture [3]) on a parallel cluster in dedicated mode and also with all processors of the same speed. The loop scheduling schemes are implemented using the Message Passing Interface on a cluster of processors. Experimental results show that the

978-0-7695-4773-2/12 $26.00 © 2012 IEEE
DOI 10.1109/NCA.2012.36

IEEE computer society

two-dimensional scheduling schemes were found to significantly reduce the total execution time of tasks over the one-dimensional schemes. In addition, the two-dimensional schemes present a more balanced load distribution of the workload among the computers in the cluster.

The rest of the paper is organized as follows. In Section II, we present some well known 1-D schemes. In Section III, we present the methodology for two-dimensional schemes. In Section IV, we present the implementation details of the scheduling schemes, present the experimental results, and compare the performance of the two-dimensional schemes with that of the one-dimensional schemes. Conclusions are derived and future work is presented in Section V.

II. One-Dimensional Loop Scheduling Schemes

In this section, we review some previously studied simple one-dimensional (1-D) dynamic loop scheduling schemes. These loop scheduling schemes were implemented using a Master-Worker architecture model [3]. In a generic dynamic (self-scheduling) scheme, at the i-th scheduling step, the master computes the chunk-size C_i (a few consecutive iterations), a starting (iteration) index $istart$, and the remaining number of tasks (iterations) R_i as follows.

Initially, $R_0 = I$ (where I denotes the total number of iterations of a parallel loop), $istart = J$ (where J denotes the lower bound of the loop). The master computes the chunk-size for the i-th scheduling step as:

$$C_i = f(R_{i-1}, p), \qquad (1)$$

where p is the number of processors. The function $f(.,.)$ can possibly have more inputs than just R_{i-1} and p. Then the master assigns to a worker processor (PE) C_i tasks and a starting (iteration) index $istart$. Then the $istart$ and R_i for the next scheduling step are updated:

$$istart = istart + C_i, \qquad R_i = R_{i-1} - C_i. \qquad (2)$$

When the user or the system has chosen a 'threshold' last chunk-size L, then the computation of C_i must be modified by adding: If ($C_i < L$) then $C_i = L$.

The different ways to compute C_i has given rise to different scheduling schemes. Below is a description of some simple one-dimensional loop scheduling schemes. These schemes have been studied and extended in [1][2][3] and references therein.

Trapezoid Self-Scheduling (TSS-1-D): $C_i = C_{i-1} - D$, with (chunk) decrement: $D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor$, where: the first and last chunk-sizes (F, L) are user/compiler-input or (by default) $F = \left\lfloor \frac{I}{2p} \right\rfloor$, and $L = 1$. The number of scheduling steps assigned: $N = \left\lceil \frac{2*I}{(F+L)} \right\rceil$.

Factoring Self-Scheduling (FSS-1-D): FSS consists of rounds of p scheduling steps. In each round i_r the master distributes $\lceil R_{i_r-1}/2 \rceil$ iterations to the p workers. Thus, $C_{p*i_r+n} = \lceil R_{i_r-1}/2p \rceil$, for $n = 1, \ldots, p$ and the remaining iterations are $R_{i_r} = R_{i_r-1}/2$.

Trapezoid Factoring Self-Scheduling (TFSS-1-D): TFSS-1-D is a scheme which uses stages (as in FSS-1-D). The size of the next chunk is the sum of the next p chunks that would have been computed by the TSS-1-D algorithm. The chunk is then equally divided among the p processors, as in FSS-1-D. Thus the TFSS-1-D chunk-size is computed as:

$$C_j^{TFSS-1-D} = \sum_{i=k}^{k+p} C_i^{FSS-1-D}$$

Guided Self-Scheduling (GSS-1-D): $C_i = \lceil R_{i-1}/p \rceil$. This is a dynamic scheme with a non-linear chunk-size function. It assigns large chunks initially, which implies reduced communication/scheduling overheads in the first scheduling steps. A modified version GSS-1-D(l) with minimum assigned chunk-size l attempts to improve on the weaknesses of GSS-1-D.

The load imbalance depends on the execution time difference between t_j, for $j = 1, \ldots, p$ where t_j is the execution time of processor j to finish all the tasks assigned to it by the scheduling scheme. This difference may be large if the first chunk is too large.

III. Two-Dimensional Loop Scheduling Schemes

In this section, we review the methodology for two-dimensional (2-D) loop scheduling schemes introduced in [14]. A 2-D scheme is derived by applying a 1-D scheme in each dimension of the loop.

Let n $(= 1, 2)$ denote the dimension of the loop and let i_n denote the index of the scheduling step and N^n denote the number of scheduling steps in dimension n. $C_{i_n}^n$ and $R_{i_n}^n$ denote the chunk-sizes and the number of remaining iterations in each dimension respectively. The chunks of the 2-D scheme will be rectangular with sizes $C_{i_1}^1 \text{x} C_{i_2}^2$ at scheduling step (i_1, i_2). Let $(istart1, istart2)$ denote the origin of the rectangular chunk at scheduling step (i_1, i_2). The 2-D schemes are derived by applying the 1-D schemes with p PEs in each loop and construct 2-D chunks which are the Cartesian products of the 1-D chunks.

Let J_1 and J_2 denote the two loops in a 2-D nested loop construct and J_n^1 and J_n^2 denote the lower and upper loop bound for the $n-th$ loop. The computation of the chunk-size and starting index point (using eqs. (1) - (2)) are performed in 2 dimensions (*i.e.* loops: J_1 and J_2) to compute $C_{i_1}^1$ and $C_{i_2}^2$ (for $i_1,\ i_2 = 1, 2, \ldots$). Initially, $istart1 = J_1^1$ and $istart2 = J_2^1$. The chunks are rectangular regions of the index space (i_1, i_2) of $N^1 \times N^2$ scheduling steps. Thus, each chunk is a rectangle which has as its origin the index point $(istart1, istart2)$ and a width and a height: $C_{i_1}^1$ and $C_{i_2}^2$, respectively. A 2-D (or rectangular) chunk is referred by its width and height *i.e.* $C_{i_1}^1 \times C_{i_2}^2$.

The rectangular chunks are computed in order along 'wavefront diagonals' (please see Figure 3 in [14]). These diagonals of rectangles in the index space start from the bottom left (index point (J_1^1, J_2^1)) and end up at the top right of the region (index point (J_1^2, J_2^2)). The starting index point is updated as follows: (1) $istart1 = istart1 + C_{i_1}^1$ or $istart1 = istart1 - C_{i_1}^1$ and (2) $istart2 = istart2 + C_{i_2}^2$ or $istart2 = istart2 - C_{i_2}^2$, because the rectangular chunks are computed along the wavefront diagonals. Based on the above, a generic 2-D loop scheduling algorithm is as follows:

A 2-D Loop Scheduling Algorithm:

Master:

- Receive a new request from a worker for tasks.
- If (*there exist unassigned rectangular chunks*) then
  - Compute a new task (2-D chunk $C_{i_1}^1 \times C_{i_2}^2$ and $(istart1, istart2)$) along the wavefront diagonals.
  - Send the new task to the worker.

  Else
  - Send a 'terminate' signal to (requesting) workers.

Worker:

- Send a request to the Master.
- Receive new tasks or a 'terminate' signal.
- Perform tasks or terminate.

An algorithm for distributed TSS-2-D, known as DTSS-2-D is presented in [14]. Here, we also implemented the DFSS-2-D.

IV. Implementation and Results

The following simple and distributed loop scheduling schemes are implemented: TSS-1-D, TSS-2-D, FSS-1-D, FSS-2-D, TFSS-1-D, TFSS-2-D, GSS-1-D, GSS-2-D, DFSS-1-D, and DFSS-2-D. The above schemes are implemented in C++ using the distributed programming framework offered by the Message Passing Interface (MPI). The computer cluster used is the Sun Constellation Linux Cluster, named Ranger at the Texas Advanced Computing Center (TACC) at the University of Texas at Austin. Each node on the Ranger system is comprised of four AMD Opteron Quad-Core 64-bit processors (16 cores in all). In our runs, we used a number p of processors/workers (cores) (with $p = 1, \ldots, 64$). All Ranger nodes are interconnected using InfiniBand technology in a full-CLOS topology providing a 1GB/sec point-to-point bandwidth.

The test problem used is the Mandelbrot computation [3]. The Mandelbrot computation is a doubly nested loop without any dependencies. The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit. We use, in our tests, the Mandelbrot fractal computation algorithm on the domain [-2.0, 2.0] × [-2.0, 2.0], for different window sizes (problem sizes/matrix sizes) (8000 × 8000 to 32000 × 32000). The algorithm uses irregular loops with unpredictable computation cost per iteration.

In the following, we present the experimental results. T_p denotes the total execution time (for a given problem size) measured on the master PE and T_{comp} denotes the computation time of a worker PE (for executing all the chunks assigned to it). All timings are in seconds (sec). The communication times of the workers in all the schemes are very small. This low value in communication time is because the workers do not send the computed results back to the master. They only send a request for work and receive a reply. These messages are small in size. We note that $T_p \approx max\{T_{comp_1}, T_{comp_2}, \ldots, T_{comp_p}\}$.

Figures 1 and 2 present the *maximum difference* (which is a measure of the imbalance of the workers computational load) in workers computation times for various 1-D and 2-D schemes with 16 and 32 PEs for a problem size of 16000 × 16000. It can be observed that the differences in the case of 1-D schemes are quite substantial and hence there is considerable load imbalance among the worker PEs. The differences in the case of 2-D schemes are small (less than 1) and hence there is almost perfect load balancing among the worker PEs.

Figures 3 and 4 present the T_p for various 1-D and 2-D simple schemes with various number of PEs and 8000 × 8000 and 16000 × 16000 problem sizes. The figures also present the results for the distributed schemes (DFSS-1-D and DFSS-2-D). It can be observed that the 2-D schemes show substantial performance improvement over the 1-D schemes. For example: (i) the T_p of FSS-2-D and GSS-2-D is more than 50% less than that of FSS-1-D and GSS-1-D for a problem size of 8000 × 8000 with 8 PEs; (ii) the 2-D schemes show around at least 40% performance improvement over the 1-D schemes for a problem size of 16000 × 16000 with 8 PEs; (iii) Performance improvements of around 75% by the DFSS-2-D over DFSS-1-D can be observed.

Table I presents the times of various 1-D and 2-D schemes with 64 workers for a problem size of 32000 × 32000. Substantial performance improvement achieved by the 2-D schemes can be observed. In the case of TSS-2-D and GSS-2-D, this is about 50%.

Figure 5 shows the total execution time (T_p) for TSS-1-D, TSS-2-D, FSS-1-D, and FSS-2-D with increasing problem size (up to 32000 × 32000). The number of worker PEs is fixed to 16. It can be observed that for all the problem sizes, the 2-D schemes show superior performance compared to the 1-D schemes. The performance of the other 2-D schemes is similar.

Figures 6 and 7 present the speedup of the various 1-D and 2-D schemes with increasing number of workers for a problem size of 16000 × 16000. We computed the speedup (S_p) according to the equation: $S_p = t_1/t_p$ where t_1 is the execution time using one PE and t_p is the execution

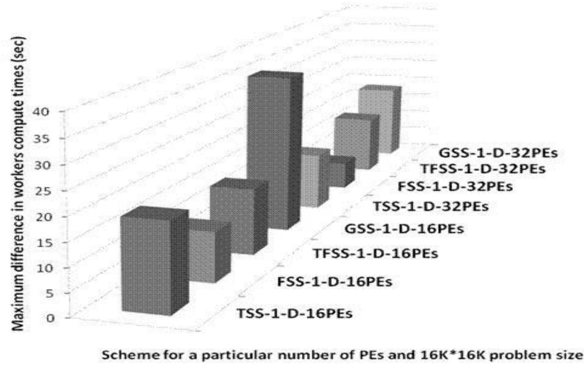

Figure 1. Maximum difference in workers compute times for various 1-D schemes with various number of PEs and 16000 × 16000 problem size

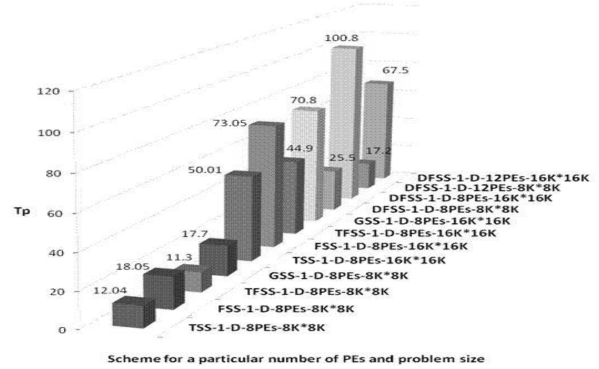

Figure 3. T_p for various 1-D schemes with various number of PEs and problem sizes

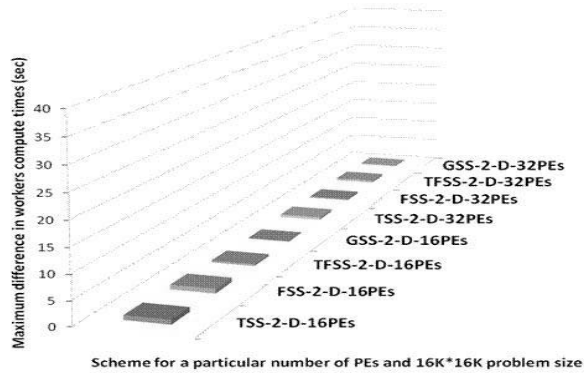

Figure 2. Maximum difference in workers compute times for various 2-D schemes with various number of PEs and 16000 × 16000 problem size

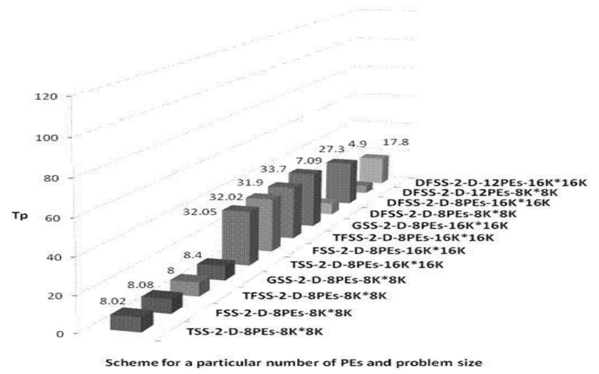

Figure 4. T_p for various 2-D schemes with various number of PEs and problem sizes

Table I
TOTAL EXECUTION TIMES (IN SECONDS) OF VARIOUS 1-D AND 2-D SCHEMES FOR 64 WORKERS AND A PROBLEM SIZE OF 32000 × 32000

PE	TSS	FSS	GSS
1-D	40.1	43.9	56.4
2-D	20.9	34.9	21.5

time using p PEs. It can be observed that, as the number of worker PEs increases, the speedup of all the schemes improves which shows that the schemes are scalable. It can also be observed that the speed up of the 2-D schemes is substantially higher than that of the 1-D schemes.

V. CONCLUSIONS

In this paper, we implemented two-dimensional loop scheduling schemes and compared their performance with that of one-dimensional loop scheduling schemes. The loop scheduling schemes are implemented using the Message Passing Interface on the Ranger cluster at the Texas Advanced Computing Center. The Mandelbrot computation is used as the test problem with size ranging from 8000 × 8000 to 32000 × 32000. Experiments were conducted with the number of processors ranging from 1 to 64. Results showed that the two-dimensional scheduling schemes perform better compared to the one-dimensional schemes and also present a more balanced load distribution of the workload among the computers in the cluster.

In future, we plan to use more number of processors with larger problem sizes to test the scalability of the schemes and also consider problems which have loops with dependencies.

ACKNOWLEDGMENTS

We gratefully recognize the following: (1) TACC of University of Texas at Austin, for providing access to the Ranger multiprocessor cluster; (2) support by NSF grant (HRD-0932339) to the University of Texas at San Antonio; (3) the reviewers comments that helped improve the quality of the paper.

REFERENCES

[1] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring," *Cluster Computing*, vol. 6, pp. 215–226, 2003.

[2] A. Kejariwal, A. Nicolau, and C. Polychronopoulos, "History-aware self-scheduling," in *International Conference on Parallel Processing*, Columbus OH, Aug 2006, pp. 185–192.

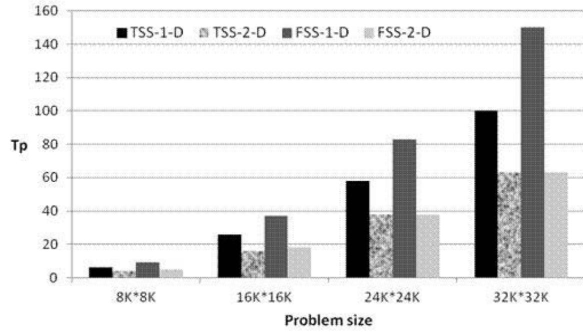

Figure 5. Total execution time of various schemes for various problem sizes. Number of PEs = 16.

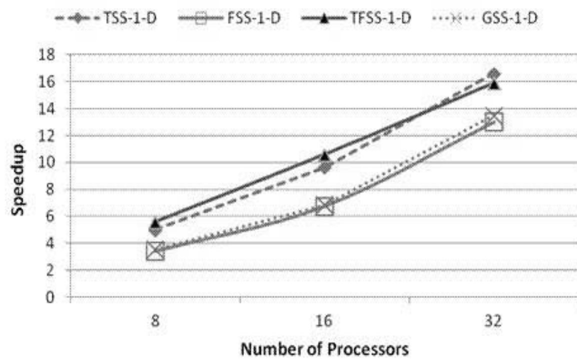

Figure 6. Speedup of various 1-D schemes with number of processors. Problem size = 16000 $\times$ 16000.

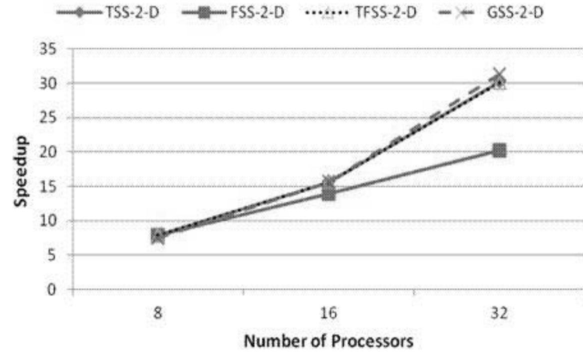

Figure 7. Speedup of various 2-D schemes with number of processors. Problem size = 16000 $\times$ 16000.

[3] A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali, "Distributed loop-scheduling schemes for heterogeneous computer systems," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 7, pp. 771–785, 2006.

[4] A. T. Chronopoulos, S. Penmatsa, N. Yu, and D. Yu, "Scalable loop self-scheduling schemes for heterogeneous clusters," *Intl. Jrnl. of Computational Science and Engineering*, vol. 1, no. 2/3/4, pp. 110–117, 2005.

[5] F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos, "Enhancing self-scheduling algoritms via synchronization and weighting," *Journal of Parallel and Distributed Computing*, vol. 68, no. 2, pp. 246–264, 2008.

[6] J. Herrera, E. Huedo, R. S. Montero, and I. M. Llorente, "Loosely-coupled loop scheduling in computational grids," in *Proc. of the 20th IEEE Intl. Parallel and Distributed Processing Symp.*, Rhodes Island, Greece, 2529 April 2006.

[7] S. Penmatsa, A. T. Chronopoulos, N. T. Karonis, and B. Toonen, "Implementation of distributed loop scheduling schemes on the teragrid," in *Proc. of the 21st IEEE Intl. Parallel and Distributed Processing Symp.*, Long Beach, California, March 2007.

[8] S. Fujita, "Semi-dynamic multiprocessor scheduling with an asymptotically optimal performance ratio," *IEICE Transactions on Fundamentals of Electronics. Communications and Computer Sciences*, vol. E92.A, no. 8, p. 17641770, 2009.

[9] J. Diaz, S. Reyes, A. Nino, and C. Munoz-Caro, "Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: Application to internet-based grids of computers," *Future Generation Computer Systems, Elsevier Publishers*, vol. 25, no. 6, p. 617626, 2009.

[10] W. C. Shih, S. S. Tseng, and C. T. Yang, "Performance study of parallel programming on cloud computing environments using mapreduce." *Information Science and Applications (ICISA)*, pp. 1–8, 2010.

[11] C. T. Yang, C. C. Wu, and J. H. Chang, "Performance-based parallel loop self-scheduling using hybrid openmp and mpi programming on multicore smp clusters." *Concurrency and Computation-Practice and Experience*, vol. 23, no. 8, p. 721744, 2011.

[12] P. Li, Q. Zhu, Q. Ji, and X. Zhu, "An approach of chunk-based task runtime prediction for self-scheduling on multi-core desk grid," *Journal of Computers*, vol. 6, no. 7, p. 13391345, 2011.

[13] C. C. Wu, C. T. Yang, K. C. Lai, and P. H. Chiu, "Designing parallel loop self-scheduling schemes using the hybrid mpi and openmp programming model for multi-core grid systems." *The Journal of Supercomputing*, vol. 59, pp. 42–60, 2012.

[14] A. T. Chronopoulos, L. M. Ni, and S. Penmatsa, "Multi-dimensional dynamic loop scheduling algorithms," in *IEEE International Conference on Cluster Computing*, Austin, TX, 17-20 Sept. 2007, pp. 241 – 248.