JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 20, 263-273 (2004)

Short Paper

A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters*

CHAO-TUNG YANG AND SHUN-CHYI CHANG
High-Performance Computing Laboratory
Department of Computer Science and Information Engineering
Tunghai University
Taichung, 407 Taiwan

Cluster computers are a viable and less expensive alternative to symmetric multiprocessor systems. However, a serious difficulty in concurrent programming of a cluster computer system is how to deal with scheduling and load balancing of such a system which may consist of heterogeneous computers. Self-scheduling schemes suitable for parallel loops with independent iterations on heterogeneous computer clusters have been designed in the past. These schemes, such as *FSS*, *GSS* and *TSS*, can achieve load balancing in SMP, even in a moderate heterogeneous environment, but are not suitable in extremely heterogeneous environments. In this paper, we propose a heuristic approach to solve parallel loop scheduling problems on an extremely heterogeneous PC cluster environment.

Keywords: parallel loops, self-scheduling, PC clusters, heterogeneous, message passing

1. INTRODUCTION

Parallel computers are becoming increasingly widespread, and nowadays many of these parallel computers are no longer shared-memory multiprocessors, but rather follow the distributed memory model for scalability. These systems may consist of homogeneous workstations, where all the workstations have processors with exactly the same specifications and identical memory and caches. However, increasingly systems are now composed of a number of heterogeneous workstations clustered together, where each workstation may have CPUs with different performance capabilities and different amounts of memory and caches, and even different architectures and operating systems.

To exploit the potential computing power of cluster computers, an important issue is how to assign tasks to computers so that the computer loads are well balanced. The problem is how to assign the different parts of a parallel application to the computing resources to minimize the overall computing time and to efficiently use the resources. An

Received January 31, 2003; accepted July 4, 2003.
Communicated by Ruay-Shiung Chang.
* A preliminary version of this paper has been presented at the 2002 International Computer Symposium, 2002.

efficient approach to extracting potential parallelism is to concentrate on the parallelism available in the loops. Since the body of a loop may be executed many times, loops often comprise a large portion of a program's parallelism. By definition, a loop is called a DOALL loop if there is no cross-iteration dependence in the loop; i.e., all the iterations of the loop can be executed in parallel. If all the iterations of a DOALL loop are distributed among different processors evenly, a high degree of parallelism can be achieved. Parallel loop scheduling is a method that attempts to evenly schedule a DOALL loop on multiprocessor systems.

According to Moore's "law" says that the number of transistors on a chip double every 18 months. But this is no scientific law, just a prediction based on past experiences and with no guarantee that it will continue this way into the future, (so says Gordon Moore,) in 18 months and so far, this law still works today. We may have to build clusters consisting of extremely different computer performance. In a homogeneous environment, workload can be partitioned equally to each working computer, but in a heterogeneous environment, this method will not work. Some researches were proposed to solve parallel loop scheduling problems on heterogeneous cluster environments by using self-scheduling schemes. These schemes will work well in a moderate heterogeneous cluster environment but not in extremely heterogeneous environments where the performance difference between the fastest computer and the slowest computer is large.

In this paper, we will propose a loop scheduling based on a self-scheduling approach to approach load balancing on extremely heterogeneous clusters. The experimental results are conducted on a PC Cluster with six nodes and where the fastest computer is 7.5 times faster than the slowest one in CPU clock speed. In our experiments, we assign 80% workload corresponding to the CPU clock, and 20% workload using traditional self-scheduling to achieve a good load balance.

The rest of the paper is organized as follows. In section 2, a brief overview of self-scheduling is given. Section 3 states our approach and reports the experiments. Finally, conclusions are given in section 4.

2. BACKGROUND

Loops are one of the most important sources of concurrency in parallel/distributed computations. In a parallel process system, two kinds of parallel loop scheduling decisions can be made, either static at compile-time or dynamic at run-time.

Static scheduling is usually applied to uniformly distributed iterations on processors [6]. However, it has the drawback of creating load imbalances when the loop style is not uniformly distributed, when the loop bounds cannot be known at compile-time; or when locality management cannot be exercised. In contrast, dynamic scheduling is more appropriate for load balancing, however, the runtime overhead must be taken into consideration. In general, parallelizing compilers distribute loop iterations by using only one kind of scheduling algorithm, which maybe static or dynamic. However, a program may have different loop styles, such as uniform workload, increasing workload, decreasing workload, or random workload.

2.1 Static Scheduling

Traditional static scheduling [6] makes a scheduling decision at compile-time and uniformly distributes loop iterations onto processors. It is applied when each loop iteration takes roughly the same amount of time, and the compiler knows how many iterations will be run and how many processors available for use at compile-time. It has the advantage of incurring the minimum scheduling overhead, but load imbalances may occur. These static scheduling schemes include Block Scheduling, Cyclic Scheduling, Block-D Scheduling, Cyclic-D Scheduling [6], etc. But these scheduling schemes are unsuitable in heterogeneous environment.

Theoretically, workload can be partitioned based on computer performance. In heterogeneous system, it is important to evaluate each computer performance, but this is not easy. Intuitively, CPU clock speed may be a good evaluation value, but it seems not enough. Many factors affect computer performance, such as the performance capability of the CPU, the amount of memory available, the cost of memory accesses, the communication medium between processors, etc [5]. Bohn and Lamont try to evaluate the performance of the computer in compiler-time [4]. In their experiment, HINT is a good benchmark. It evaluates processor and memory performance for any data type and returns a single value, “QUIPS”. Bohn and Lamont declared “QUIPS” can represent the computer performance. It has the advantage that all computers execute a part of the program - no control computer is needed. But, HINT requires hours to execute, meaning this way will not scale well. It takes a long time to add one more computer and if we want to change the peripheral, for example to replace RAM from PC-100 to PC-133, we might have to rerun HINT.

2.2 Dynamic Scheduling

Dynamic scheduling adjusts the schedule during execution whenever it is uncertain how many iterations will be needed or when each iteration will take a different amount of time due to a branching statement inside the loop. Although it is more suitable for load balancing between processors, runtime overhead and memory contention must be considered.

Self-scheduling is a large class of adaptive/dynamic centralized loop scheduling schemes. We will study these schemes from the perspective of distributed systems. For this, we use the master-slave architecture model-idle slave PCs communicate requests to the master for new loop iterations.

The number of iterations a PC should be assigned is an important issue. Due to PC’s heterogeneity and communication overhead, assigning the wrong PC a large number of iterations at the wrong time may cause load imbalancing. Also, assigning a small number of iterations may cause too much communication and scheduling overhead. Although dynamic scheduling may achieve load balancing, a master computer which will be responsible for assigning subtask to slave is needed. The master computer is not responsible for workload.

In a generic self-scheduling scheme, at the i-th scheduling step, the master computers the chunk-size is C_i and the remaining number of tasks is R_i,

$$R_0 = I,\ C_i = f(R_{i-1}, p),\ R_i = R_{i-1} - C_i$$

where $f()$ is a function possibly of more inputs than just R_{i-1} and p. Then the master assigns C_i tasks to a slave PC. A master/slave model for loop scheduling is shown in Fig. 1. Imbalancing depends on the execution time gap between t_j, for $j = 1, \ldots, p$. 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 [7].

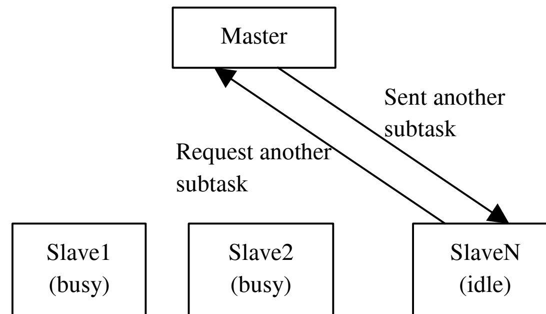

Fig. 1. A master/slave model.

The different ways to compute C_i has given rise to different scheduling schemes. The most notable examples are the following.

Pure Self-Scheduling (*SS*) This is the easiest and most straightforward dynamic loop scheduling algorithm [8]. Whenever a processor is idle, an iteration is allocated to it. This algorithm achieves good load balancing but also introduces excessive overhead.

Chunk Self-Scheduling (*CSS*) Instead of allocating one iteration to an idle processor as in self-scheduling, *CSS* allocates k iterations each time, where k, called the chunk size, is fixed and must be specified by either the programmer or the compiler [8]. When the chunk size is one, this scheme is pure self-scheduling. If the chunk size is set to the bound of the parallel loop equally divided by the number of processors, the scheme becomes static scheduling. A large chunk size will cause load imbalancing while a small chunk is likely to produce too much scheduling overhead. For different partitioning schemes, we adopted *CSS*(s), which is a modified version of *CSS*, where s means the size of chunks.

Guided Self-Scheduling (*GSS*) This algorithm can dynamically change the number of iterations assigned to each processor [2]. More specifically, the next chunk size is determined by dividing the number of remaining iterations of a parallel loop by the number of available processors. The property of decreasing chunk size implies an effort is made to achieve load balancing and to reduce the scheduling overhead. By allocating large chunks at the beginning of a parallel loop, one can reduce the frequency of mutually exclusive accesses to shared loop indices. The small chunks at the end of a loop partition serve to balance the workload across all the processors.

Factoring In some cases *GSS* might assign too much work to the first few processors, so that the remaining iterations are not time-consuming enough to balance the workload.

This situation arises when the initial iterations in a loop are much more time-consuming than later iterations. The factoring algorithm addresses this problem [1]. The allocation of loop iterations to processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Because Factoring allocates a subset of the remaining iterations in each phase, it balances loads better than *GSS* when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of Factoring is not significantly larger than that of *GSS*.

Trapezoid Self-Scheduling (*TSS*) This approach tries to reduce the need for synchronization while still maintaining a reasonable load balance [3]. *TSS*($\mathcal{N}_s$, $\mathcal{N}_f$) assigns the first Ns iterations of a loop to the processor starting the loop and the last $\mathcal{N}_f$ iterations to the processor performing the last fetch, where N_s and N_f are both specified by either the programmer or the parallelizing compiler. This algorithm allocates large chunks of iterations to the first few processors and successively smaller chunks to the last few processors. Tzen and Ni proposed *TSS*(*N/2P*, 1) as a general selection. In this case, the first chunk is of size $\frac{N}{2p}$, and consecutive chunks differ in size by $\frac{N}{8p^2}$ iterations. The difference in the size of successive chunks is always a constant in *TSS* whereas it is a decreasing function in *GSS* and in Factoring.

Intelligent Parallel Loop Scheduling (*IPLS*) Fann, Yang, Tseng and Tsai proposed a knowledge-based approach to solving loop-scheduling problems. A rule-based system, called *IPLS*, was developed by combining a repertory grid and an attribute ordering table to construct a knowledge base. *IPLS* chooses an appropriate scheduling algorithm by inferring some features of loops and assigning parallel loops to multiprocessors to achieve significant speedup [9].

Table 1 shows the different chunk sizes for a problem with the number of iteration $I = 1000$ and the number of processor $p = 4$.

Table 1. Sample partition sizes.

Scheme	Partition size
PSS	1,1,1,1,1,1,1…
CSS(125)	125,125,125,125,125,125,125,125
FSS	125,125,125,125,63,63,63,63,31,…
GSS	250,188,141,106,79,59,45,33,25,…
TSS	125,117,109,101,93,85,77,69,61,…

3. PROPOSED APPROACH

In extremely heterogeneous environment, cluster computers have extremely different performance. In this situation, additional slave computers may not get good performance because these known self-scheduling schemes partition the size of loop iterations according to a formula instead of computer performance. In *FSS*, for example,

every slave gets a size of $N/2p$ workload, where N is the total workload; p is the number of processors. If the performance difference between the fastest computer and the slowest computer is larger than $N/2p$, then load imbalance happens. Furthermore, dynamic load balancing should not be aware of the run-time behavior of the applications before execution. But in *GSS* and *TSS*, to achieve good performance, computer performance has to be ordered in an extremely heterogeneous environment.

A combination of machine types is used to test the behavior of these techniques in a heterogeneous computing environment, and matrix multiplication is chosen as the test application to get a heuristic result due to its regular behavior.

This experiment included four computers. One of them is assigned as master using *TSS*. The master is a PC with a 300MHz CPU and 208MB physical memory. The three slaves are PCs, with 1.6GHz CPU and 256MB physical memory, 233MHz CPU and 96MB physical memory, and 200MHz CPU and 64MB physical memory. The slaves are added sequentially in this order. We use *TSS* and *FSS* to test matrix multiplication with different problem sizes of 512 × 512, 1024 × 1024, and 2048 × 2048 by floating point operations. Table 2 shows our experimental result. Note that just one slave in Table 2 means that all work is done by the fastest computer only. We can see that the performance of two slave computers is less than the performance of one high speed slave computer.

Table 2. The result performance of number of slaves in an extremely heterogeneous environment.

No. of slaves	Execution time (*TSS*)			Execution time (*FSS*)		
	512 × 512	1024 × 1024	2048 × 2048	512 × 512	1024 × 1024	2048 × 2048
1	0'12"066	1'44"357	17'12"483	0'12"136	1'44"688	17'11"402
2	0'17"520	2'49"652	19'34"016	0'18"371	3'16"561	23'48"723
3	0'13"339	1'53"202	16'30"651	0'14"543	2'00"491	16'36"007

As mentioned above, in heterogeneous environment, intuitively, we may want to partition problem size according to CPU clock speed. However, the CPU clock is not the only factor which affects computer performance. Many other factors also have dramatic influences in this respect, such as the amount of memory available, the cost of memory accesses, and the communication medium between processors, etc. Using this intuitive approach, the result will be degraded if the performance prediction is accurate. A computer with the most inaccurate prediction, being the last one to finish the assigned job, is called the dominate computer.

We propose to partition α% of workload according to the performance weighted by CPU clock speed and the (100-α)% of workload according to the known self-scheduling schemes. To get load balancing, we make the (100-α)% of workload wait the dominate computer until it finishes its job. Using this approach, we don't need to know the real computer performance. The computer which finishes its job early gets a larger job to wait for the slower computer. Another advantage of using this approach is the reduction of communication. When this approach is applied to the matrix multiplication experiment, with $\alpha = 80$, a better performance is obtained.

Loops can be roughly divided into four kinds as shown in Fig. 2, uniform workload, increasing workload, decreasing workload, and random workload loops. They are the most common ones in programs, and should cover most cases. These four kinds can be classified two types, predictable and unpredictable. Our approach is suitable in all applications with predictable loops.

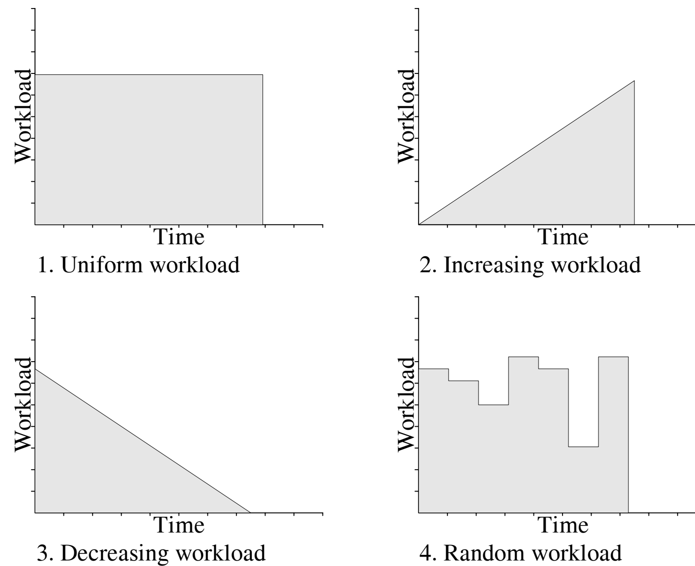

Fig. 2. Four kinds of loops.

Algorithms MASTER and SLAVE in pseudo code:
Module MASTER

```
/* performs task scheduling and load balancing */
Initialization
Gather CPU clock speed of all slave computers
r = 0;
For (i = 1; i < number_slave; i++) {
    Partition α% of loop iteration corresponding CPU clock speed and sent data to slave
r++;
}
Partition (100-α)% of loop iteration into task queue using some known self-scheduling
scheme
Probe for some data
Do {
    Distinguish source and receive data
    If task queue not empty
      Send other data to this idle slave
      r--;
    Else
```

```
      Send TAG = 0 to this idle slave
} while (r > 0);
Finalization
END MASTER

Module SLAVE /* worker */
Initialization
Send my CPU clock speed to master
Probe if some data in
While (TAG > 0) {
   Receive initial solution and size of subtask work and compute to find solution
   Send the result to master
   Probe if some data in
}
Finalization
END SLAVE
```

The approach is applied in an extremely heterogeneous environment which includes six computers. One of them is assigned as the master. The master is a PC with a 300 MHz CPU and physical memory of 208MB. Two of the slaves are PCs with 200 MHz CPUs and physical memory of 64MB. The other three slaves are PCs, with a 233 MHz CPU and physical memory of 96MB, 533MHz CPU and physical memory of 128MB, and a 1.6GHz CPU and physical memory of 256MB. These computers may own various NIC and cost of memory access, regarding as part of computer performance. SWAP occurs in some computers. If not serious, this will not affect the result.

The parameter α should not be too small or too big. In the first case, the dominate computer will not finish its work and lead to bad performance. In the second case, the dynamic scheduling strategy is rigid. In both cases, good performance can not be attained. An appropriate α value will lead to good performance and reduce communication times. Many α values are applied to the experiments, and $\alpha = 80$ result in the best performance.

Table 3 and Fig. 3 show the result in $\alpha = 80$. The column name "None" stands for "no load-balancing" and workload be partitioned just by CPU clock. "*FSS*/80" stand for "$\alpha = 80$, and remainder use *FSS* to partition" and so on. In our case, using this approach for 2048 × 2048 matrix multiplication will get 25%, 30%, 21% performance improvement for *FSS*, *GSS*, and *TSS* respectively. Note that in an extremely heterogeneous environment, known self-scheduling gets worse performance than schemes partitioning workload merely according to the CPU clock speed.

Table 3. The result of our approach in an extremely heterogeneous environment ($\alpha = 80$).

	None	*FSS*	*FSS*/80	*GSS*	*GSS*/80	*TSS*	*TSS*/80
512 × 512	8.1	9.8	7.0	10.0	7.5	9.1	7.6
1024 × 1024	74.9	98.7	56.6	115.5	59.4	71.6	63.0
2048 × 2048	598.6	678.1	509.3	732.6	509.0	666.1	521.3

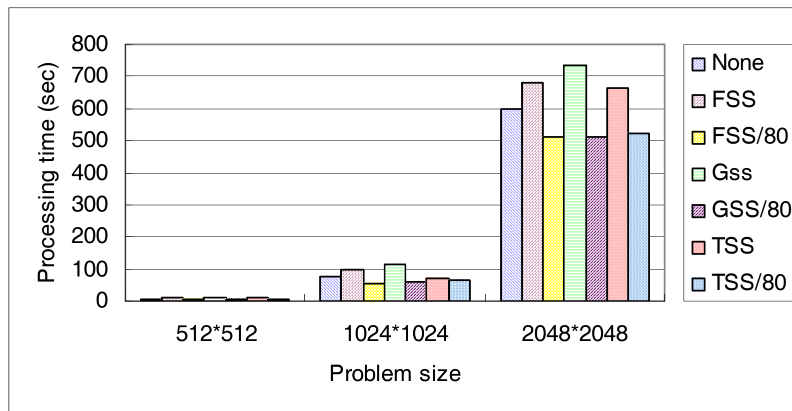

Fig. 3. The results of our approach in an extremely heterogeneous environment ($\alpha = 80$).

Our approach is also useful in a moderately heterogeneous environment. Following experiment includes five computers. One of them is assigned as the master. The master is a PC with a 900MHz CPU and physical memory of 256MB. Two slaves are PCs with 600MHz CPU speed and physical memory of 256MB. The other two slaves are PCs, with 975MHz CPU and physical memory of 512MB, 900MHz CPU and physical memory of 256MB. Table 4 and Fig. 4 present the results when the approach is applied to matrix multiplication, as $\alpha = 80$. It shows that our approach has equal or better performance than the known self-scheduling.

Table 4. The results of our approach in a moderately heterogeneous environment ($\alpha = 80$).

	None	*FSS*	*FSS*/80	*GSS*	*GSS*/80	*TSS*
1024 × 1024	64.6	59.1	59.1	59.1	59.0	65.2
2048 × 2048	520.8	477.3	474.4	477.6	474.3	505.8
3072 × 3072	1792.8	1720.9	1711.0	1715.8	1714.7	1792.1

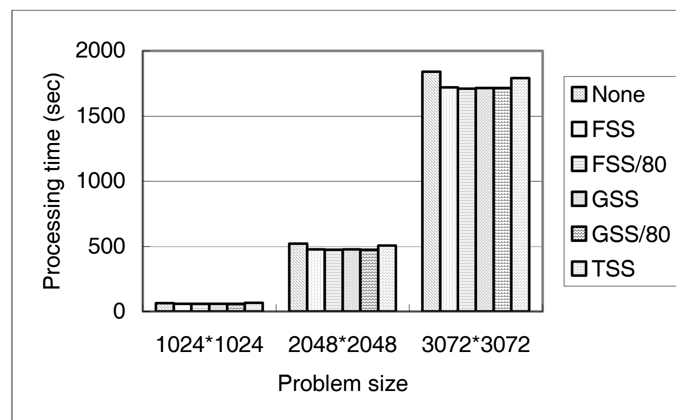

Fig. 4. The results of our approach in a moderately heterogeneous environment ($\alpha = 80$).

4. CONCLUSIONS AND FUTURE WORK

In extremely heterogeneous environments, known self-scheduling schemes cannot achieve good load balancing. In this paper, we propose an approach to partition loop iterations and achieve good performance in such environments. We propose that partitioning 80% of the workload according to the performance weighted by CPU clock speed and the 20% of workload according to known self-scheduling. In the case of matrix multiplication with 2048×2048 problem size, our approach can obtain more 30% performance improvement than *GSS*. Therefore, our approach is suitable in all applications with predictable loops. In the near future, we will solve parallel loop scheduling problems with unpredictable loops on extremely heterogeneous PC clusters.

REFERENCES

1. S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: a method scheme for scheduling parallel loops," *Communications of the ACM*, Vol. 35, 1992, pp. 90-101.
2. C. D. Polychronopoulos and D. Kuck, "Guided self-scheduling: a practical scheduling scheme for parallel supercomputers," *IEEE Transactions on Computers*, Vol. 36, 1987, pp. 1425-1439.
3. T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, 1993, pp. 87-98.
4. C. A. Bohn and G. B. Lamont, "Load balancing for heterogeneous clusters of PCs," *Future Generation Computer Systems*, Vol. 18, 2002, pp. 389-400.
5. E. Post and H. A. Goosen, "Evaluating the parallel performance of a heterogeneous system," in *Proceedings of 5th International Conference and Exhibition on High-Performance Computing in the Asia-Pacific Region* (*HPC Asia 2001*), http://parallel.hpc.unsw.edu.au/HPCAsia/papers/12.pdf.
6. H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on NUMA multiprocessors," in *Proceedings of International Conference on Parallel Processing*, Vol. II, 1993, pp. 140-147.
7. A. T. Chronopoulos, R. Andonie, M. Benche, and D. Grosu, "A class of loop self-scheduling for heterogeneous clusters," in *Proceedings of 3rd IEEE International Conference on Cluster Computing* (*CLUSTER 2001*), 2001, pp. 282-291.
8. P. Tang and P. C. Yew, "Processor self-scheduling for multiple-nested parallel loops," in *Proceedings of International Conference on Parallel Processing* (*ICPP '86*), 1986, pp. 528-535.
9. Y.-W. Fann, C.-T. Yang, S.-S. Tseng, and C.-J. Tsai, "An intelligent parallel loop scheduling for multiprocessor systems," *Journal of Information Science and Engineering-Special Issue on Parallel and Distributed Computing*, Vol. 16, 2000, pp. 169-200.

Chao-Tung Yang (楊朝棟) was born on November 9, 1968 in I-Lan, Taiwan, R.O.C. He had received a B.S. degree in Computer Science and Information Engineering from Tunghai University in 1990 and an M.S. degree in Computer and Information Science from National Chiao Tung University in 1992. He received Ph.D. degree in Computer and Information Science at National Chiao Tung University in July 1996. He had passed the first class of the National Higher Examination in Information Processing field in 1994. He was the winner of 1996 Acer Dragon Award for outstanding Ph.D. Dissertation. Since 1996, he has worked as an associate researcher for ground operations at ROCSAT Ground System Section (RGS) of National Space Program Office (NSPO) in Hsinchu Science and Industry Park. Since Aug. 2001, he has been on the faculty of the Department of Computer Science and Information Engineering at Tunghai University, Taichung, Taiwan, and currently is an associate professor. His current research interests were in parallel and cluster computing, grid computing and parallelizing compilers.

Shun-Chyi Chang (張順奇) was born on September 13, 1971 in Taichung, Taiwan, R.O.C. He received the M.S. degree in 2003 from the Department of Computer Science and Information Engineering at Tunghai University. Now, he works at Taichung Office, Bureau of Consular Affairs, Ministry of Foreign Affairs, R.O.C. His research interests include parallel processing, database design and bioinformatics.