Parallelized computation for computer simulation of electrocardiograms using personal computers with multi-core CPU and general-purpose GPU

Wenfeng Shen\textsuperscript{a,b}, Daming Wei\textsuperscript{a,*}, Weimin Xu\textsuperscript{b}, Xin Zhu\textsuperscript{a}, Shizhong Yuan\textsuperscript{a,b}

\textsuperscript{a} Biomedical Information Technology Lab, The University of Aizu, Uegami 90, Tsuruga, Ikki-machi, Aizu-Wakamatsu, Fukushima 965-8580, Japan
\textsuperscript{b} School of Computer Engineering and Science, Shanghai University, Yanchang Road, Shanghai 200072, China

Article info

Article history:
Received 21 March 2010
Received in revised form 8 June 2010
Accepted 25 June 2010

Keywords:
Computer simulation
Whole-heart model
General-purpose GPU
CUDA
Load-prediction dynamic scheduling

Abstract

Biological computations like electrocardiological modelling and simulation usually require high-performance computing environments. This paper introduces an implementation of parallel computation for computer simulation of electrocardiograms (ECGs) in a personal computer environment with an Intel CPU of Core (TM) 2 Quad Q6600 and a GPU of Geforce 8800GT, with software support by OpenMP and CUDA. It was tested in three parallelization device setups: (a) a four-core CPU without a general-purpose GPU, (b) a general-purpose GPU plus 1 core of CPU, and (c) a four-core CPU plus a general-purpose GPU. To effectively take advantage of a multi-core CPU and a general-purpose GPU, an algorithm based on load-prediction dynamic scheduling was developed and applied to setting (c). In the simulation with 1600 time steps, the speedup of the parallel computation as compared to the serial computation was 3.9 in setting (a), 16.8 in setting (b), and 20.0 in setting (c). This study demonstrates that a current PC with a multi-core CPU and a general-purpose GPU provides a good environment for parallel computations in biological modelling and simulation studies.

1. Introduction

Computer simulation of an electrocardiogram (ECG) based on a whole-heart model is an important topic in the study of the bioelectric phenomena of the heart [1]. In general, the computational burden for this type of study is considerably heavy. For this reason, parallel computations were considered in previous studies. So far, most parallel algorithms for the computer simulation of ECGs were designed for running on supercomputers or UNIX workstations with multiple processors. For example, Tudem et al. reported a parallel algorithm for the simulation of QRST integral maps with a membrane-based computer heart model using a Silicon Graphic Origin 2000 computer with 64 processors [2]. Reumann et al. realized large-scale cardiac modelling on the IBM Blue Gene/L parallel computers with 16,384 computational nodes [3]. Zhu et al. described a computer simulation of an ECG based on a 3D cardiac model on a 4-node SUN computer cluster using OpenMP and MPI protocols [4]. Recently, some researchers have also realized the computer simulation of an ECG using Intel Xeon-based clusters. Ten Tusscher and Panfilov conducted their simulations on a basis of the model of the ventricular conduction system using MPI on a Beowulf cluster consisting of 10 Dell 650 Precision workstations with 20 dual...
Intel Xeon 2.66 GHz central processing units (CPUs) [5]. Lu et al. conducted the multi-scale modelling of rodent ventricular myocytes using an Intel Xeon-based cluster with 10 processors [6].

The ideas and works introduced above for parallelizing the computations in an electrocardiological computer simulation are based on large-scale and high-performance computing environments. However, these systems are more expensive and difficult to manage than personal computers (PCs). It would be ideal if personal computers could be used as computational platforms for the parallelization of computations in cardiac simulation studies. In fact, the rapid progress in PC hardware has made this technically possible. For example, based on such an idea, Sato et al. employed general-purpose computing on a general-purpose Graphics Processing Unit (GPU) to accelerate the simulations of electrical wave propagation in cardiac tissues and obtained a speedup of 30 as compared to the computation with a single 2.0 GHz AMD Opteron processor for 2D tissue simulations [7]. They found that the computational speed using a single general-purpose GPU for a 3D anatomical heart simulation was 1.6 times slower than that using a 32-CPU Opteron cluster and that the cluster with 2 or 4 general-purpose GPUs was faster than the same cluster without using general-purpose GPUs. Obviously, the general-purpose GPU with many cores can greatly increase the computation power of a PC.

Our research group has already developed a whole-heart model for the computer simulation of the body surface ECG potentials [8]. This model has been recently updated for the computer simulation of intracardiac ECG potentials [9]. The main computational burden in our simulation lies in solving numerical boundary problems governed by a Poisson’s equation, which yields the body surface ECG potentials at 684 triangles, epicardial surface ECGs at 2002 triangles, and endocardial surface ECG potentials at 1162 triangles. It took more than 11 h in our experiment to conduct a 4.8-s-long simulation at a 3-ms interval through using a PC with a single core CPU.

Nowadays, a typical PC is usually equipped with a multi-core CPU and a general-purpose GPU, supported by development software such as OpenMP API and Compute Unified Device Architecture (CUDA). This environment provides us with a good platform to parallelize computations in computer simulation. In this paper, we report our latest results related to the parallelization of computations in the simulation of an ECG based on our whole-heart model. We propose a load-prediction dynamic scheduling scheme to achieve efficient scheduling on the hybrid environment. In the following sections, we first summarize CUDA and OpenMP API, and then analyze the serial algorithm for computing the ECG potentials based on the whole-heart model and describe the parallelization algorithms in three parallelization device setups: (a) a four-core CPU without a general-purpose GPU, (b) a general-purpose GPU plus 1 core of CPU, and (c) a four-core CPU plus a general-purpose GPU. We finally report our experiments and results to show how these algorithms significantly accelerated the computation of a computer simulation in an efficient and economical way.

2. CUDA and OpenMP

2.1. CUDA

Recently, the computing capacities of GPUs have improved exponentially. Based on the single-instruction multiple-data (SIMD) structure, GPUs are suitable for intensive computing and capable of other applications besides graphics computations, which they were originally designed for. Therefore, a new concept, general-purpose GPU has been devised and stands for general-purpose computation on GPUs.

Compute Unified Device Architecture (CUDA) includes a set of library functions as extensions of the C language [10]. CUDA can offload data-parallel and compute-intensive tasks to NVIDIA GPUs, where the computation is distributed in a grid of GPU thread blocks. All blocks are appointed GPU threads that execute a program, i.e. a kernel. Each block is identified by a 2-dimension block ID, and each GPU thread within a block can be identified by a GPU thread ID for easy indexing of the data to be processed. The block and grid dimensions known as the execution configurations can be set runtime and are typically based on the size and dimension of the data to be processed. NVIDIA GPUs have a memory hierarchy with five kinds of memory hardware i.e. shared memory, constant memory, texture memory, global memory, and local memory. The access time varies for different types of memories. This enhancement in the memory model allows programmers to better exploit the parallel capacity of a GPU for general-purpose computing [11].

A typical CUDA implementation consists of the following stages:

1. Allocate data on the device.
2. Transfer data from the host memory to the device memory.
3. Initialize device memory if necessary.
4. Determine the execution configuration of GPU.
5. Execute kernels and store the results in the device memory.
6. Transfer the calculation results from the device memory to the host memory [12].

2.2. OpenMP

The OpenMP Application Programming Interface (API) is a standard for parallel programming on shared-memory multiprocessors. OpenMP is efficient at describing loop-level parallelism in applications, which is a target for utilizing multi-core CPU’s highly parallel computing units to accelerate data-parallel computations, i.e. OpenMP provides a set of directives that allow the user to annotate a sequential program to indicate how it should be executed in parallel. The parallel for directive creates a team of CPU threads and distributes the iterations of the related “for” loop among these CPU threads. Then the CPU threads execute the appointed subsets of the iterations in parallel. After completing the tasks, the CPU threads suspend except CPU thread 0 [13].
The serial algorithm for the computer simulation of ECGs includes three major steps: the simulation of the excitation propagation in the heart model based on the propagation model of the Huygens’s type, the computation of the ECG potentials in the volume conductor, and the storage of the results. The second step takes most of the total processing time because it involves the calculation of the potential on approximately 3800 triangles of four boundaries, including the torso surface and the three surfaces of ventricular muscles of the heart model produced by a maximum of approximately 50,000 electric current dipole sources. With the hypothesis that each triangle element in the volume conductor model is of iso-potential, we employed the boundary element method to calculate the simulated body surface ECGs and epicardial and endocardial ECGs from the simulated current dipole sources on the basis of the following equation:

\[
\phi(r_{kj}) = \frac{1}{2\pi}\sum_{i=1}^{M} J(\tau) \cdot \frac{r_{kj} - r_{\tau}}{|r_{kj} - r_{\tau}|^3} + \frac{1}{2\pi} \sum_{l=1}^{N} \sum_{m=1}^{T_{l}} (\sigma_{l} - \sigma_{l}^+) \cdot \phi(r_{lm}) \Delta \Omega_{k/l,m},
\]

where \(\phi(r_{kj})\) and \(r_{kj}\) are the potential and central location vector on the \(j\)-th triangle of the \(k\)-th surface, \(\sigma_{k}\) is the electric conductivity, the superscripts \((-\) and \((+)\) are the inside and outside signs of the boundary surfaces, \(M\) is the total number of the cell models’ current dipole sources, \(J\) is the current dipole source at the cell model, \(r_{\tau}\) is the location of the \(i\)-th current dipole source, \(T_{l}\) is the number of triangles on the \(i\)-th surface \((l=1-4, \text{corresponding to the 1st-4th surfaces, the number of surfaces } N=4)\), and \(\Delta \Omega_{k/l,m}\) is the solid angle subtended by the triangle element at \(r_{lm}\) viewed from \(r_{kj}\). In the volume conductor model, the body surface, the epicardial surface, the right endocardial surface, and the left endocardial surface were divided into 344, 1002, 307, and 278 nodes, and 684, 2002, 610, and 552 triangles [14]. Then, (1) was written as follows [9]:

\[
\phi(r_{kj}) = b_{kj} + \sum_{l=1}^{4} g_{kj,l} \Phi_{l},
\]

where:

\[
b_{kj} = \frac{1}{2\pi} \sum_{i=1}^{M} J(\tau) \cdot \frac{r_{kj} - r_{\tau}}{|r_{kj} - r_{\tau}|^3},
\]

\[
g_{kj,l} = \frac{(\sigma_{l} - \sigma_{l}^+)}{2\pi(\sigma_{l} - \sigma_{l}^+)} \frac{1}{\Delta \Omega_{k/l,m}} \left( \Delta \Omega_{k/l,m} \right). \quad k \neq l,
\]

\[
g_{kj,k} = \frac{(\sigma_{k} - \sigma_{k}^+)}{2\pi(\sigma_{k} - \sigma_{k}^+)} \left( \Delta \Omega_{k/l,m} \right) \times \Delta \Omega_{k/l,m}.
\]

As A and its inverse were calculated in advance, the computation of an ECG depended mainly on the calculation of \(b_{kj}\) in (2), i.e. \(B\) in (4), and resolving the linear equation in (4). The computation procedure is described with pseudo codes in Fig. 1, where Propagation() is the function used for conducting the simulation of excitation propagation and calculating the dipoles in the heart, \(N\) is the number of time steps in the simulation, ECG_in() is the function used for calculating \(b_{kj}\) in (2), and ECG_b() is the function used for calculating the ECG potentials in (4). Fig. 2 illustrates the definition of ECG_in() with pseudo codes, where \(i, j, K\) are the maximum coordinate values of the cell models in three dimensions. When the computation complexity of \(b_{kj}\) for a single dipole in (2) was \(N_{i}\), the maximum number of current dipole sources was \(N_d\), and the total number of the triangles on the volume conductor’s surfaces was \(N_{t}\), the total computation complexity for calculating all \(b_{kj}\) in the \(N\) steps was \(O(N_{i}N_{d}N_{t}).\) Further, the computation complexity for (4) was \(O(N_{t}^2N)\) when the inverse matrix of \(A\) was calculated in advance. We carried out an experiment and obtained the total computational

\[
\Phi_{i} = (\Phi_{i,1}, \ldots, \Phi_{i,K})^T.
\]

\[
b_{k} = (b_{k,1}, \ldots, b_{k,K})^T \quad \text{and} \quad G_{k,l} = (g_{k,l,1}, \ldots, g_{k,l,K})^T.
\]

\[
A \Phi = B.
\]
time for each function. Propagation() took less than 0.1% of the total computational time, ECG_inf() took approximately 99%, ECG_bd() took approximately 0.7%, and SaveResults() took approximately 0.2% of the total computational time. Propagation() and SaveResults() placed a light computation burden. Therefore, the parallelization for the computer simulation of ECGs in our study focused on the calculation of ECGs.

4. Parallelization of computer simulation of ECGs

ECG_inf() and ECG_bd() are executed iteratively and therefore feasible for parallelization based on a multi-core CPU and a general-purpose GPU. To compare the performance of the parallelization on the basis of the different hardware configurations available for a PC, we paralleled the computation of the ECG simulation using three parallelization device setups: (a) a four-core CPU without a general-purpose GPU, (b) a general-purpose GPU plus 1 core of CPU, and (c) a four-core CPU plus a general-purpose GPU.

4.1. Parallelization using OpenMP based on multi-core CPU

OpenMP has been employed for developing parallel algorithms using a multi-core CPU. Many load balancing strategies have been proposed for heterogeneous cluster systems [16–19], while few researches have been conducted on task scheduling for the parallel computation on a hybrid structure PC with a multi-core CPU and a general-purpose GPU. Generally, there are two types of scheduling strategies: static and dynamic scheduling [15]. The advantage of static scheduling is that it is easy to be implemented and has no extra overhead for allocating the tasks at runtime. However, if the computation procedure is of load unbalancing, static scheduling may decrease the efficiency of parallel computing. In dynamic scheduling, the task is allocated to a core runtime, i.e. in the course of job execution. The advantage of dynamic scheduling is that it can maintain a load-balancing runtime. Further, the main time cost factors are the runtime overheads for the task allocation synchronization. As illustrated in Fig. 2, the computational complexity in a time step depends on the number of excited cardiac cell models. As the differences in the computational complexity between the adjacent time steps are small, the computation loads of the CPU cores are nearly equal and balanced if we allocate the tasks using static scheduling, which is easy to realize efficiently.

The pseudo code of the main program for parallel computation based on OpenMP is given in Fig. 3. The OpenMP directive ‘#pragma omp parallel’ defines a parallel program section and initializes the multi-thread, which executes the code of the program section. The variables ‘tid’ and ‘tnum’ are private variables of the CPU thread id and the number of CPU threads. The two OpenMP functions ‘omp_get_thread_num()’ and ‘omp_get_num_threads()’ return the ID of the currently executing CPU thread and the number of CPU threads in the currently executing parallelized computation. The private variable ‘i’ is initialized as the ‘tid’ of the CPU thread. At the end of each iteration, ‘i’ will increase ‘tnum’. Therefore, in the successive iteration, the CPU thread will start from the ‘i + tnum’ task. The static scheduling strategy assigns computational tasks in terms of the numbers of iterations and does not require synchronization; therefore, it can be easily realized and can work efficiently. We have compared the computation efficiencies of the algorithm based on both static and dynamic scheduling with an Intel(R) Core(TM2) Quad CPU Q6600. The parallel program based on static scheduling takes 10121.21 s, while the parallel program based on dynamic scheduling takes 10163.13 s for the same computation. Therefore, in this case, the algorithm based on static scheduling is more efficient than that based on dynamic scheduling, i.e. the saved time from the increased efficiency obtained by dynamic scheduling cannot compensate for the time cost for the runtime overheads.

4.2. Parallelization using CUDA based on a general-purpose GPU

As for the parallelization based on a general-purpose GPU, we developed it using the CUDA’s programming model, which involved the usages of a kernel, GPU thread hierarchy, and memory hierarchy. Figs. 4 and 5 illustrate the pseudo codes for the parallel computation using CUDA. The kernel is a program
to be executed in parallel by GPU threads. In our algorithm, the kernel is the sub-function of GPUECG\_inf\_single(), i.e. the function for calculating the contribution of one dipole to the $b_{k,j}$ value in (2). As shown in Fig. 5, GPUECG\_inf\_single() called the kernel program to calculate the $b_{k,j}$ value because of a single dipole on the body surface (684 triangles), the left endocardial surface (552 triangles), the right endocardial surface (610 triangles), and epicardium (2002 triangles). CUDA automatically appointed each GPU thread to a core on the general-purpose GPU for parallel calculation. As for the memory operation, we used the GPUMalloc() function to request the memory for saving the dipoles and ECG data in the device memory. The MemcpyHostToDevice() function transferred the data from the host memory to the device memory. It should be noted that all the necessary data were transmitted from the host memory to the device memory at the beginning of the computation to save transmission time. Then, we employed the DeviceMeminit() function to initialize the arrays in the device memory, as shown in Fig. 5, in order to save the calculation results in the device memory. At the end of function ECG\_inf(), the MemcpyDeviceToHost() function transferred the computational results from the device memory to the host memory. During this parallelization, we only used one core of the CPU for data transmission and GPU configurations.

4.3. Parallelization using CUDA and OpenMP based on a general-purpose GPU and a multi-core CPU

As OpenMP and CUDA can fully exploit multi-core CPUs and general-purpose GPUs, respectively, to speed up the computation, we expect to accelerate the computer simulation of an ECG by combining a multi-core CPU and a general-purpose GPU.

In this study, OpenMP initialized four CPU threads, which were automatically allocated to the four cores of the CPU for parallel computation. As a co-processor of the CPU, the general-purpose GPU was appointed to speed up the computation of CPU thread 0. Therefore, the calculation performance of CPU thread 0 is faster for high-intensity computations and slower for low-intensity ones as compared with other CPU threads. Correspondingly, we used the dynamic scheduling to realize the load balancing in this hybrid computing platform.

(1) Dynamic scheduling

Figs. 6 and 7 illustrate the pseudo codes for the parallel computation based on dynamic scheduling. In Fig. 6, CPU
(2) Load-prediction dynamic scheduling

During the realization of load balancing using dynamic scheduling in the parallel computation, we found that dynamic scheduling could not fully exploit the computational performance of the general-purpose GPU. As shown in Fig. 8, the computation time of each time step in the serial program was unequal. The computation burdens of some time steps were rather low, and therefore, CPU thread 0 could not fully exploit the computational performance of the general-purpose GPU when it executed these light computation tasks.

In this paper, we propose a load-prediction dynamic scheduling method to solve the problem mentioned above. First, by predicting the computational time using the number of the excited cardiac cell models in each time step, we sorted the time steps according to the predicted computational time in the descending order. As the general-purpose GPU has many cores for computation, the time steps with heavy computation burden should be allocated to be executed on the GPU, and the time steps with light computation burden should be allocated to the CPU threads except CPU thread 0. Therefore, the tasks (time steps) were dynamically allotted according to the CPU thread IDs from the two directions of the deque, an array with the sorted time step. As shown in Fig. 9, the ‘Subscript’, ‘Head’, and ‘Tail’ represent the locations of the task allotted to threads in the deque, the head of the deque, and the tail of deque, respectively. CPU thread 0 was allocated the heavy tasks from thread 0 executes the CUDA program. Further, the ‘pragma omp critical’ code controls the read and write operations of the global variable ‘i’ and allocates tasks to each idle CPU thread.

```
1   if (tid==0)
2     {
3       DeviceMemInit();
4     }
5   else
6     {
7       ArrayInit();
8     }
9   for k=0 to K
10      for j=0 to J
11        for i=0 to I
12          if (dipole(i, j, k)==0)
13            if (tid==0)
14              
15                GPUECG_inf_single();
16            }
17          else
18            
19                ECG_inf_single();
20       
21   if (tid==0) MemoryDeviceToHost();
```

Fig. 7 – ECG_inf() function program pseudo code using CUDA and OpenMP.

Fig. 8 – Computing time for ECG_inf() in each time step.

Fig. 9 – Example of deque.
the head of the deque and then configured the GPU to execute these tasks, and the other CPU threads were allocated the light tasks from the tail of the deque.

As the time cost of the ECG\text{inf}() functions accounted for 99% of the total computing time, we only sorted the computational time for ECG\text{inf}() in all time steps. Through the analysis discussed earlier, we found that the computing time in each time step was directly proportional to the number of excited cardiac cell models in the time step. Therefore, the computational time in each time step could be predicted from the simulation results of the activation propagation. As shown in Fig. 10, the ‘LoadPredictandSort(deque)’ predicted the computational time in each time step and saved the time step number after sorting it to the deque. The ‘pragma omp critical’ realized the dynamic task allocation. The global variables ‘Count’, ‘Head’, and ‘Tail’ contained the allotted time step numbers; these variables were initialized with 1, the head, and the tail of the deque, respectively. The ‘Subscript’ and ‘myCount’ were the private variables for saving the location of the task in the deque and the count of the time steps allotted to a CPU thread, respectively.

5. Experiments and results

To evaluate the parallel algorithm, we used a PC equipped with an Intel(R) Core(TM2) Quad CPU Q6600, 3 GB host memory, and a GeForce 8800 GT card as the evaluation platform. The GeForce 8800 GT card has 14 streaming multiprocessors, each of which has 8 scalar processors and a 512 MB device memory on the card. The serial and parallel algorithms were employed for the reproduction of a 4.8-s ECG during the induction and sustaining of supraventricular tachycardia using programmed electrical stimulation. First, we invoked the serial program in the machine and saved the simulation results and the elapsed time. Then, we invoked the parallel algorithm on the same machine and saved the results and the elapsed time.

Through a comparison, we concluded that the calculated ECGs of parallel computation using OpenMP were equal to those of the serial computation. There was a small difference (relative error = 1.0 × e\text{-4}) in the results of parallel computing if CUDA was used. The difference was attributed to the float32 multiplication and division operations in the general-purpose GPU.

The time cost of the serial calculation using one core of the CPU was 39760.21 s, and the computing time for the parallel program using OpenMP based on static and dynamic scheduling was 10121.21 s and 10163.13 s, respectively. The time cost in parallel computing using a general-purpose GPU plus a core of CPU was 2363.29 s. Figs. 12 and 13 illustrate the speedup of the parallelization using both CUDA and OpenMP.

6. Discussion

6.1. Parallelization using OpenMP

Although the efficiency of static scheduling was higher, the difference in the cost time using the two scheduling methods was small for the parallelization using only a multi-core CPU.

Fig. 10 – Pseudo code of main program based on load-prediction dynamic scheduling.
As shown in Fig. 8, the number of excited cardiac cell models in the simulation of activation propagation changed gradually. Consequently, the computational complexity between the adjacent computation time steps was small, and the computational burden of each core in the multi-core CPU was similar. Furthermore, the time cost for the runtime overheads and synchronization during dynamic scheduling counteracted the time saved because of the increased efficiency; hence, the parallel program based on dynamic scheduling took more computation time than that based on static scheduling. Therefore, static scheduling was sufficient for the parallelization using OpenMP and could be realized more easily as an extra control such as runtime overheads; thus, synchronization among threads was unnecessary.

The speedup has been defined in (5), where $T_s$ is the computation time of the serial program and $T_p$ is the computation time of the parallel program.

$$\text{speedup} = \frac{T_s}{T_p}$$

Fig. 11 demonstrates the computing time and speedup for different numbers of CPU threads using OpenMP and static scheduling. The speedup increased with an increase in the number of CPU threads. When the number of CPU threads was four, the speedup reached 3.92. This indicated that the algorithm had excellent scalability. Therefore, we could improve the computing speed by increasing the number of CPU threads using a CPU with more cores.

6.2. Parallelization using CUDA

The speedup of parallelization using a general-purpose GPU was increased to 16.8. There were two factors contributing to the higher speedup value: (1) the highly efficient kernel program and (2) the efficient use of the storage system of the general-purpose GPU.

It was demonstrated that the performance of parallel computing based on the general-purpose GPU was apparently considerably higher than that of the 4-core CPU used in our evaluation platform.

6.3. Parallelization based on CUDA and OpenMP

Fig. 12 illustrates the speedup of parallelization using CUDA and OpenMP for 1600 time steps and different number of CPU threads. As the parallel computation based on CUDA and OpenMP could take complete advantage of the computational capacities of both the general-purpose GPU and the multi-core CPU through the dynamic task scheduling strategy, the speedup increased from 16.8, when only the general-purpose
GPU was used, to 20.0. Furthermore, when only a single CPU thread was used, the speedup based on load-prediction dynamic scheduling was smaller than that based on dynamic scheduling because of the overhead for load prediction and sorting. When the number of CPU threads was increased to 2 or more than 2, the speedup was approximately 1–1.5 higher for the parallel computation based on load-prediction dynamic scheduling compared with that based on dynamic scheduling. Fig. 13 illustrates the speedup of parallelization using CUDA and OpenMP for 4 CPU threads and different number of time steps. The speedup was higher for the parallel computation based on load-prediction dynamic scheduling compared with that based on dynamic scheduling too. This is because of the ability of the load-prediction dynamic scheduling to fully exert the advantage of the general-purpose GPU, which is feasible for large-scale parallel computing.

7. Conclusion

In this research, we employed the Wei-Harumi model based on a propagation model of the Huygens’s type instead of reaction-diffusion models based on ion currents used in other researches [7] and most of the calculation burden lies in the calculation of electrocardiograms from dipole sources, whose parallel computation was introduced in this paper. We studied parallel computing with three parallelization device setups: (a) four-core CPU without a general-purpose GPU, (b) a general-purpose GPU plus a core of CPU, and (c) four-core GPU plus a general-purpose GPU. The parallel computing using a general-purpose GPU was clearly faster than that using a multi-core CPU. The parallel computation using both multi-core CPU and general-purpose GPU obtained the highest speedup 20.0 based on the load-prediction dynamic scheduling. Obviously personal computers with a multi-core CPU and a GPGPU provide us an ideal supercomputing platform. Our research extended the application of parallel computing to an ECG simulation on the personal supercomputing platform. We expect to further improve the computing performance by using multiple high-performance general-purpose GPUs in a PC in the future.

Acknowledgment

This work was supported in part by Japan Society for the Promotion of Science under Grants-In-Aid for Scientific Research No. 21500297.

References


