Creating an easy to use and high performance parallel platform on multi-cores networks

Viet Hai Ha[1], Xuan Huyen Do[2], Van Long Tran[3], and Éric Renault[3]

[1] College of Education, Hue University, Vietnam
haviethai@gmail.com
[2] College of Sciences, Hue University, Vietnam
doxuanhuyen@gmail.com
[3] SAMOVA, Télécom SudParis, CNRS, Université Paris-Saclay,
9 rue Charles Fourier - 91011 Evry Cedex, France
{van_long.tran,eric.renault}@telecom-sudparis.eu

Abstract. How to easily exploit the performance of network using multi-core processors nodes is the purpose of many researches including CAPE (Checkpointing Aided Parallel Execution). CAPE uses the checkpointing technique to bring the simplicity and high performance of OpenMP – a high performance and easy-to-use standard of parallel programming API on shared-memory architecture – onto distributed-memory architectures. Theoretical analysis and experimental results have proved that CAPE has ability of providing a high performance and complete compatibility with OpenMP standard. This article aims at introducing how to use multiple processes on calculating nodes to increase performance of CAPE with the initial results.

Keywords: CAPE, Checkpointing Aided Parallel Execution, OpenMP, Parallel Programming, Distributed Computing, HPC

1 Introduction

1.1 OpenMP

OpenMP [1] is an API providing a high level abstraction for parallel programming on shared-memory architectures. It consists of a set of environment variables, directives and functions that support easily converting sequential C/C++ or Fortran programs into parallel programs.

OpenMP uses fork-join model with thread as the basic parallel structure. Initially, the program consists of only one master thread processing sequence code. Whenever meeting an OpenMP parallel directive, this master thread spawns a team work including itself and a set of slave threads (phase fork) and tasks are divided into these threads. After the slave threads have finished their tasks, the result is updated in the memory space of the main master thread and they can finish work (phase join). So, after this phase, the program remains only one thread as the original program.

OpenMP uses a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the memory. In addition, each thread is allowed to have its own temporary view of the memory. Currently, OpenMP has only been completely installed for the shared-memory architecture because of the complexity of the installation of all the requirements of OpenMP on the other memory model. This is the motivation for many researchs to be conducted with the objective of installing OpenMP on the distributed-memory architecture. However, there is not any result having meet the two requirements of fully compatibility with OpenMP standard and high performance. The most prominent alternatives may include the use of SSI [2]; SCASH [3]; compiled into MPI [4][5]; the use of Global Array [6]; and Cluster OpenMP [7]. Even with Cluster OpenMP, a commercial product from Intel also requires the use of its own more directives (not belonged in OpenMP standard) in some cases. And therefore, it is not a fully compatible installation of OpenMP.

1.2 CAPE

Principle of CAPE: CAPE (Checkpointing Aided Parallel Execution) [8] is a new approach to install OpenMP on distributed-memory systems. CAPE uses process as the basic parallel unit, instead of using thread in original OpenMP. With CAPE, all the most important tasks of the fork-join model are automatically implemented using checkpointing technique, including the division of task to slave processes, extraction of results slaves-ones and the updating these results in the memory space of the master process. space of the master process.

Deployment model: Figure 1 illustrate CAPE deployment diagram. In there:

Master node: plays the role of master thread in the operating model of OpenMP. Accordingly, it executes the code section of the master process, distributes the job to slave processes and receives the results achieved by slave processes after completing the parallel code blocks. These tasks are performed by the modules:

- User Application: the user's program code, originally written in the original language (CAPE is supporting with C language) along with OpenMP directives. This program has been translated by the CAPE program into a standard C code and then continues to be translated into machine code by a conventional C compiler, such as gcc of GNU.
- Distributor: This is the program which sets nodes in the system and distributes tasks for nodes. Distributor basing on IP of node to distinguish the nodes and activate the program on those nodes with the corresponding parameter. Currently, the distributor is installed by a Shell program, with the input parameters are the IP of the nodes in the system as well as the role of those nodes in the operational model of CAPE.
- Monitor: this program is both a checkpointer (snapshot progress of process) and also a management application program. In checkpointer role, it is a

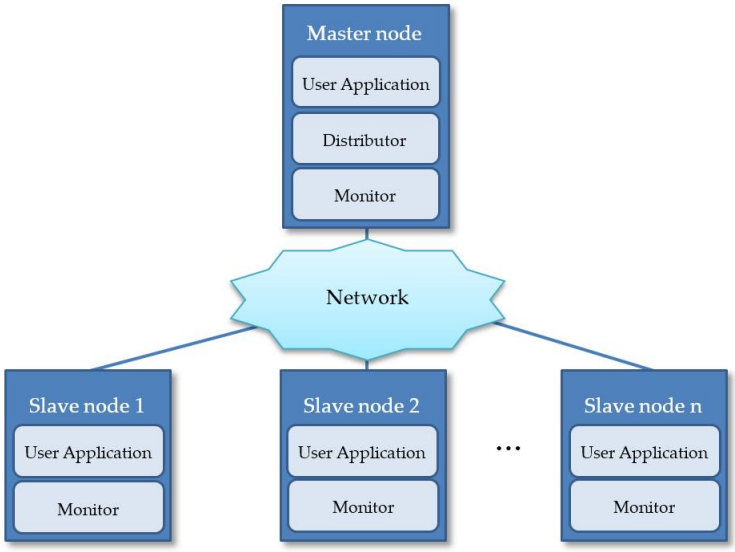

Fig. 1. CAFE deployment model

discontinuous incremental checkpointing [9], take on two main tasks are: 1) make the snapshot progress in each specific code to initialize the status for the processes at the slave nodes at the beginning point of the parallel code; and 2) extract the implementation results of the parallel sections of code in slave nodes. Absolutely, it is also responsible for updating the memory space of the process by the snapshot processes which increase discretely at the identified locations.. In the role of managing application program, the monitor is responsible for initializing application program; communicate and exchange data between nodes including allocating the tasks, and sending calculation results from the slave nodes to the master one, etc.

Slave nodes: are the nodes performing the calculations in the parallel code. At those nodes, only two modules are the user's applications, which are required to play the role of calculation; and monitor. The operating model at theses nodes is generating application program of users according to the requests sent from the distribution of master node, receiving calculation requirements, performing calculations and extracting calculation results back to the master node.

Results: CAPE has been developed and installed to achieve the parallel OpenMP directives such as parallel for, parallel section. We have carried out experiments on a square matrix multiplication, with various sizes from 3000x3000 to 12000x12000, for calculating the number of nodes varies between 2 and 30 [10]. Figure 2 shows the acceleration coefficient (speedup) of CAPE over the number of machines and different matrix sizes. The dashed lines represent the theoretical

maximum increase. This chart shows clearly that the solution CAPE achieved very good results, with the measured speed ratio is in the range of 75% to 90% in compare with the increase in the theoretical maximum acceleration coefficient. To some extent, the graph also performs the scalable (scalable) of CAPE when the speed line is almost linear, and no sign of a significant decline in the knotty calculations.

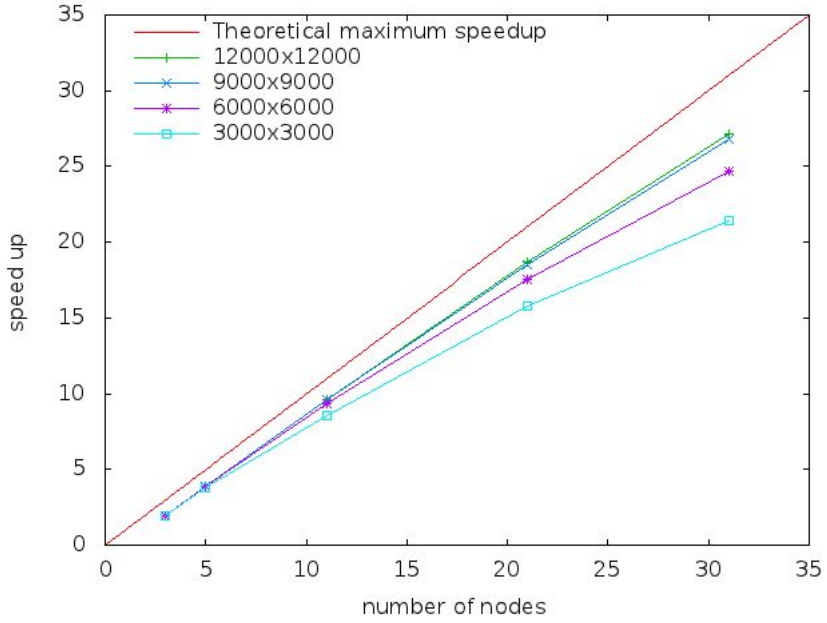

Fig. 2. Speedup of CAPE

Disadvantage on multicore systems: Multi-core processor is a single processor with two or more independent processing units, each unit can independently execute commands of programs. Nowaday, all new computers have 2 to 8 core processors; it leads to the popularity of computer networks with multicore nodes. In order to reduce calculation time, application programs can exploit the capabilities of multicore processors by running different parallel parts of programs on these cores. OpenMP is a typical example of this direction by using multithread execution model as mentioned in section 1.1.

With the current execution model of CAPE, in each calculation node, there is only one process running application programs. Moreover, this is a single process and is not divided into sub-threads. Therefore, at each calculation node, the commands of the application programs are sequentially executed. This can

be clearly seen when looking at the graph measuring the execution parameters of system while running a CAPE program, as shown in Figure 3. As seen on this chart, only the third core is fully exploited, while the other core is nearly inactive. Thus, the calculation resources of the system are being wasted and effectively exploiting them can reduce the running time of programs, i.e. increase the efficiency of their executions.

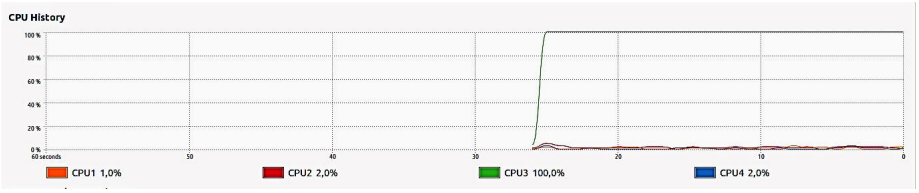

Fig. 3. Ratio of exploiting cores while running one process on each calculation node

2 Using multiple processes in calculating nodes to increase performance of cape in multi-cores systems

2.1 Principle

It should be noted that according to the execution model of CAPE, at the slave nodes, application code are sub-parts of parallel sections in original OpenMP programs. As consequence, these codes are potential in parallelization. Therefore, the general direction to effectively exploit the calculation resources in slave nodes is doing two things: 1) continue to divide the task at each node into subcomponents; and 2) make them concurrently running on different cores of processor. With multi-task operating systems today, the application program is not necessary to implement the second explicitly; it is taken over by the operating system. Meanwhile, with multi-process and multi-thread programs that use many calculation resources, the operating system will distribute calculating resources in an optimal way. For computers using multi-cores processors, the cores normally are load balancing. Thus, in the case of CAPE, only the first task has to be performed, by organizing the code in multi-process or multi-thread models. In there, multi-process model can be implemented directly by running multiple applications on a slave node, i.e. multiple processes on one physical machine. In this way, for each running time, the program will create a process and it will take a calculation part of parallel code.

2.2 Implementation details

To create multiple processes by running multiple applications on a slave node, it is necessary to modify the distributor in the CAPE deployment model so

that the application program is initialized many times with different parameters. Therefore, in each slave node, there are many application processes that performed different parts of the parallel sections. This is easily done as shown in the example below, in which the system has three nodes including one master node with IP address is 192.168.122.1 and two slave nodes with IP adresse respectively is 192.168.122.179, and 192.168.122.223. The name of application program is mulmt.

The original code of the distributor is a Shell script, when only one process of application program on each slave node is shown below (in which the ordinal number is added for the convenience of the presentation)

```
1.  #!bin/sh
2.  folder=/home/hahai/cape2/cdv9
3.  prog=mulmt
4.  num_nodes=2
5.  master=192.168.122.1
6.  node1=192.168.122.179
7.  node2=192.168.122.223
8. ${folder}/dbpf -f ${folder}/${prog} -a ${master}
    -k ${num_nodes} -o 0
9. ssh ${node1} ${folder}/dbpf -f ${folder}/${prog}
    -a ${master} -k ${num_nodes} -o 1 &
10. ssh ${node2} ${folder}/dbpf -f ${folder}/${prog}
    -a ${master} -k ${num_nodes} -o 2 &
11. exit 0
```

Explanation:

- Line 1: Specify interpreter Shell sh will be used
- Line 2: Specify the location of CAPE program
- Line 3: Specify the name of application program
- Line 4: Number of slave nodes
- Line 5: IP of the master nodes
- Line 6,7: IP of the slave nodes
- Line 8: Initialize the monitor and the application program on the master node
- Line 9,10: Initialize the monitor and the application program on the first and the second slave nodes
- Line 11: Exit Program

With the command lines above, the system will be initialized when the user execute this shell programs at the master node. Meanwhile, by the command at line 8, the monitor (dbpf) will start and it enables the application program (mulmt). The parameters of master IP address, number of slave nodes, and indicators of process are also transferred. Because the process index transferred is 0 (parameter - o 0), the application program should know it will act as the

master node in the execution model of CAPE. The command lines number 9 and 10, are executed by the remote invocation ssh with IP adress of the slave node. The other parameters are the same as in the command line number 8, except the process index is not 0 so that the application may know it is a slave node in the execution CAPE model. It can be noticed that with each slave node, there is only one call to the application program so there is only one application process is initialized. Moreover, when analyzing the conversion model of CAPE serves for transferring the OpenMP parallel constructs, it can be seen that in the slave nodes the application code is sequentially performed. Therefore, at each slave node, there is only one single process of application programs. This makes CAPE can not fully exploite the capabilities of multi-core processors, as stated in Section 1.2. In order to overcome this drawback in the way of running many times the application program on each slave node, the command lines number 9 and 10 will be cloned as shown in the code below, in which the application program is run twice on each slave node. Finally, the code of distributor is rewritten with the lines number 4 and 9, 10 are modified as below, while the other lines remain as original codes.

```
4.  num_nodes=4
...
9.  ssh ${node1} ${folder}/dbpf -f ${folder}/${prog}
   -a ${master} -k ${num_nodes} -o 1 &
9a. ssh ${node1} ${folder}/dbpf -f ${folder}/${prog}
   -a ${master} -k ${num_nodes} -o 2 &
10. ssh ${node2} ${folder}/dbpf -f ${folder}/${prog}
   -a ${master} -k ${num_nodes} -o 3&
10a.ssh ${node2} ${folder}/dbpf -f ${folder}/${prog}
   -a ${master} -k ${num_nodes} -o 4 &
```

In which:

- Line 4: is modified to have 4 slave procesesses.
- Line 9: is cloned into lines 9 and 9a, with the process index in line 9a is 2. Thus, in the first slave node, there is 2 times the application program is executed, i.e. there are 2 processes are created for the parallel code.
- Line 10: is processed in the same way of line 9, which is cloned into line 10 and line 10a, with process index is 3 and 4 respectively.

3 Experiments

To evaluate the feasibility as well as the performance of the proposed method, we have tested it with the matrix multiplication problem on a cluster with nodes equipped an Intel Core i3 processors (2 cores - 4 threads) running at 3.5GHz, 4GB RAM, uses Ubuntu 14:04, connected by 100Mb/s Ethernet. The experiments were conducted with two scenarios are varying number of processes on the slave nodes and the matrix sizes. Some of the test results are presented below.

3.1 The exploiting the capabilities of multi-core processors

Due to many application processes with high requirement of calculation resources running in concurrent, the capacity of multi-core processors are exploited better. This is clearly shown in the graph the proportion of the execution of the processors. Such as the case of 4 processes running on each slave node, the ratio of the core activities reaches 100%, as seen on the chart in Figure 4. These results are very different from the case in which there is only one application process on each slave node, when only one core is exploited with its full capacity, while the other ones are nearly inactive, as shown in Figure 3. All proved that the use of multiple processes has better exploited the performance of multi-core processors.

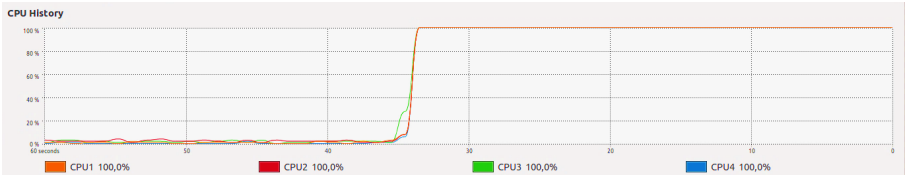

Fig. 4. Ratio of exploiting cores while running multiple processes on each slave node

3.2 Acceleration with different processes on each slave node

To evaluate this, we have tested on a system with 11 nodes (1 master node, and 10 slave nodes); with the matrix size is 6000x6000. The experiment is conducted with the number of application processes on each slave node respectively is 1, 2 and 4. Note that the case of running one application process is also the case with the old execution model of CAPE.

The chart shows that the execution time decreases when running multiple processes. For the case of running 2 processes, execution time is reduced to nearly a half in comparing with the case running one process. This is reasonable cause for each node at this section, the number of calculation commands that are the ones that consumes the most calculation resources, are reduced by a half. For the case of running 4 processes, the processing time is decreased but it is also greater than the case of 2 processes. This can be explained by the mechanism of CAPE, whenever executing an application process, it is always accompanied by a process of monitor. Therefore, in fact, when running 4 processes of application program, there are up to 8 processes implemented in parallel. Although monitor process does not take too much calculation resources, it also has a certain influence on the distribution of system resources.

On the side of the master node, the result is good while running two processes in which the execution time also reduced by approximately a half. However, in the case of using 4 processes, the time strongly increases, even higher than the

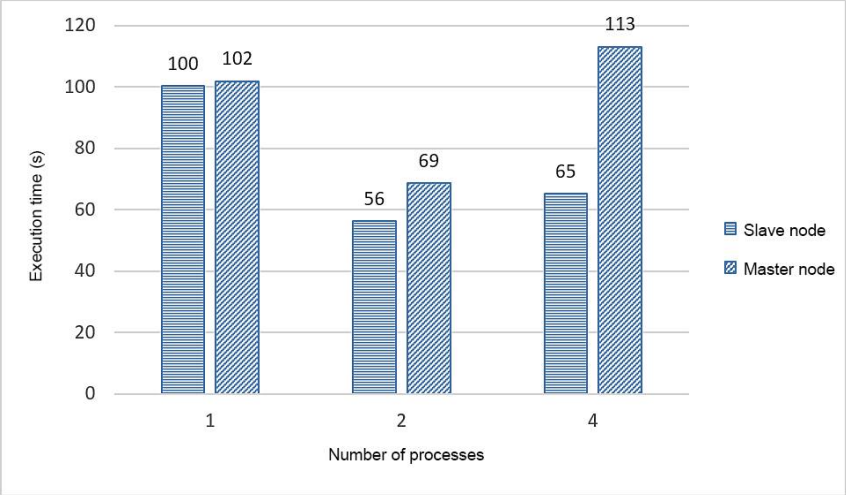

Fig. 5. Compare the execution time of program with different processes

case of using one process. This result is unreasonable if we don't analyze it carefully. Back to the results in Figure 2, when using an application process on each slave node, CAPE can generate a nearly linear speedup with the number of nodes, i.e. the number of calculation processes, including the maximum number of calculation nodes is 30. Consequently, there are two main causes of the abnormal increase of the execution time. The first is the architecture of the processors, with 2 real cores and 4 threads, instead of 4 real cores. The second is due to the multiple processes in the slave nodes have overlapping IP addresses when processing requests of setting the socket from these nodes, that causes a conflict and this needs time for solving. This is also the cause of the system failure when increasing the number of the processes on each slave node. However, this conclusion needs to be tested by conducting experiments and measurements in more details.

As shown in the Figure 5, it is also the preliminary conclusion that with the machines using 2 cores – 4 threads, the optimal number of processes is 2.

3.3 Acceleration with different problem sizes

To evaluate the scalability according to the problem sizes of multiple processes model, a similar experiment was conducted, with 6 nodes (5 slave nodes and 1 master node), using 4 processes on each slave node. The results of measurement are shown in the diagram in Figure 6. These results are accordance with the complexity of matrix multiply problem.

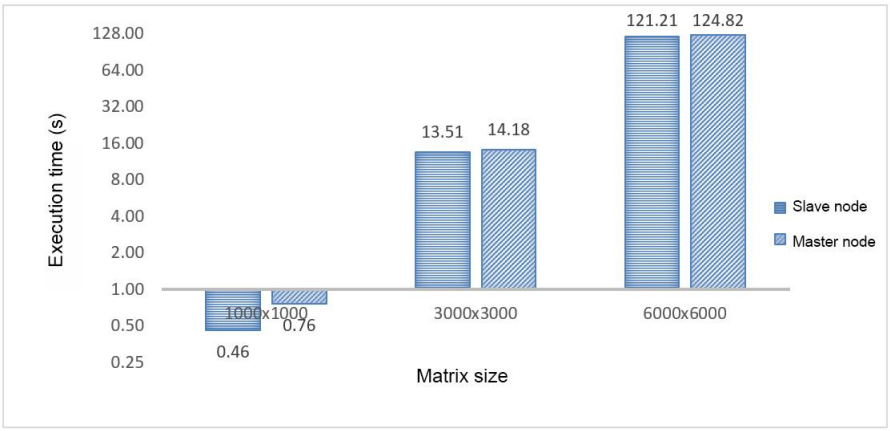

Fig. 6. Acceleration with different problem sizes

3.4 Advantages and disadvantages

The outstanding advantage of this approach is simplicity, the program codes are almost unchanged (except for editing in the distributor). The experiments showed the performance is significantly increased, nearly double in the cases using 2 processes on each slave node. Theoretically, the slave nodes using N-core processors can run N times the application program to maximum exploit the processing capabilities. However, each application process requires one monitor process so the number of application processes in optimal case is less, and in this experiment, is N/2. This is also the first drawback of this direction. The second drawback is the possibility of conflicts among the programs on the same node. Moreover, the use of socket to implement the data transmission on network can also cause conflicts over resources between the programs and cause error with the large processes numbers. Finally, the use of multiple independent processes of the same program requires the slave nodes to have big amount of RAM to ensure the speed of execution.

4 Conclution

CAFE, with the basic principles and the initial experimental results have shown its great potential to become a fully compatible implemetation and high performance for OpenMP on distributed-memory architecture. Some researchs is being continued to develop CAPE in many directions towards a fully implementation of OpenMP on these architectures, as well as in the direction of exploiting the new architecture of the processor to increase its performance. For networks using multi-core processors machines, the direction of using multiple processes on the calculation nodes by running multiple application programs has been tested and

presented in this paper. This is a simple way, without requiring many changes in CAPE programs and but provides higher performance than the case of running one process in the previous model. However, there are still many shortcomings that need to be overcome such as resource conflicts which reduces the performance and causes system errors. That is one of our directions for developing CAPE in the near future.

References

1. OpenMP, A.: Openmp application programming interface 4.5 (2015)
2. Morin, C., Lottiaux, R., Vallée, G., Gallard, P., Utard, G., Badrinath, R., Rilling, L.: Kerrighed: a single system image cluster operating system for high performance computing. In: Euro-Par 2003 Parallel Processing. Springer (2003) 1291–1294
3. Sato, M., Harada, H., Hasegawa, A., Ishikawa, Y.: Cluster-enabled openmp: An openmp compiler for the scash software distributed shared memory system. Scientific Programming **9**(2, 3) (2001) 123–130
4. Basumallik, A., Eigenmann, R.: Towards automatic translation of openmp to mpi. In: Proceedings of the 19th annual international conference on Supercomputing, ACM (2005) 189–198
5. Dorta, A.J., Badía, J.M., Quintana, E.S., de Sande, F.: Implementing openmp for clusters on top of mpi. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer (2005) 148–155
6. Huang, L., Chapman, B., Liu, Z.: Towards a more efficient implementation of openmp for clusters via translation to global arrays. Parallel Computing **31**(10) (2005) 1114–1139
7. Hoeflinger, J.P.: Extending openmp to clusters. white paper (2006)
8. Renault, É.: Distributed implementation of openmp based on checkpointing aided parallel execution. In: A Practical Programming Model for the Multi-Core Era. Springer (2007) 195–206
9. Ha, V.H., Renault, É.: Discontinuous incremental: A new approach towards extremely lightweight checkpoints. In: Computer Networks and Distributed Systems (CNDS), 2011 International Symposium on, IEEE (2011) 227–232
10. Ha, V.H., Renault, E.: Improving performance of cape using discontinuous incremental checkpointing. In: High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on, IEEE (2011) 802–807