MPI and OpenMP Paradigms on Cluster with multicores and its application on FFT

Yongjin Li
School of Information Science and Technology, Northwest University NWU
Xi'an, China
E-mail: lyj2003_02@163.com

Weichang Shen
School of Information Science and Technology, Northwest University NWU
Xi'an, China
E-mail: jzbi@nwu.edu.cn

Anlei Shi/ Lidong He/ Dong Zhao
School of Information Science and Technology, Northwest University NWU
Xi'an, China
E-mail: sailor0105@163.com

***Abstract*—At present the cluster of workstation with multicores is most popular in high performance architecture. However, pure MPI paradigm can't benefit from the computing capability of multicores. Parallel programming can combine distributed memory parallelization with shared memory parallelization. And hybrid MPI+OpenMP may be a superior solution because of its less communication, declined consumption of the memory, and improved load balance. In this paper, we propose an improved parallel FFT algorithm based on hybrid architecture and the results shows the algorithm has good scalability and high efficiency.**

Keywords- Parallel FFT; MPI; OpenMP; COW;MultiCores

I. INTRODUCTION

The cluster of workstation (COW) provides an economic solution for high-performance calculation (HPC). Nowadays, the emergency of multicores processor (Intel Core Duo, Intel Core2Duo, AMD multicores) greatly enhanced the computing capability of COW. Figure 1 shows the structure of this architecture. The multi-core shared-memory computing nodes are coupled via high-speed interconnects. Inside the node, details like UMA (Uniform Memory Access), while outside with NUMA (Non-Uniform Memory Access).

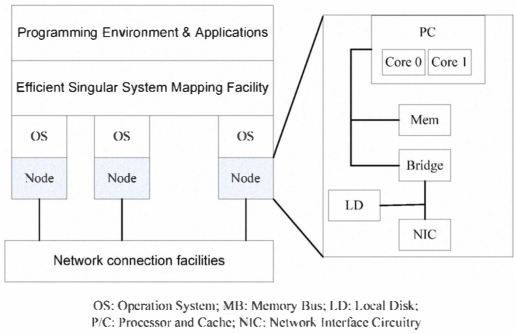

Figure 1. cluster of workstation with multicores

MPI may be the best choice for distribute system. However, we can't make good use of the multi-core. Although the MPI programming model also supports multithreads, the implementations of multithreads can not shows the superiority of the multi-core computer. And OpenMP can't be applied into distribute system. Consequently, it seems that it will be reasonable to employ a hybrid programming model, a hybrid programming model adopts OpenMP for parallelization inside the node and adopts MPI for message passing between nodes.

The hybrid programming model MPI+OpenMP are useful to solve the problems of load balancing of parallel applications independently of the architecture [1]. A hybrid MPI+OpenMP programming is currently being implemented to obtain high efficiency codes on the hybrid computer. In this paper, we propose a hard FFT algorithm base on this hybrid programming model and the hybrid FFT algorithm outperforms better than pure MPI. In this paper, we propose a parallel FFT algorithm in hybrid program which can maximize the utilization of the computing capability supplied by this architecture. And the parallel programming mode [8] used in this paper is shown in figure 2.

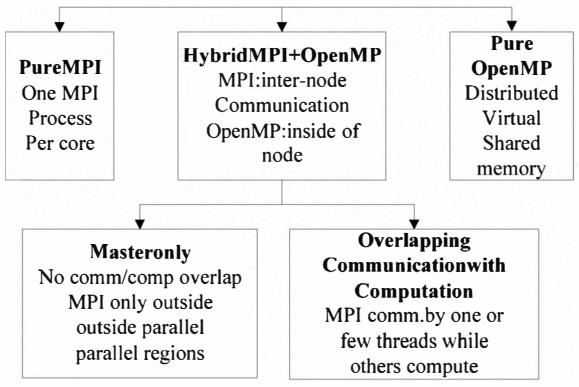

Figure 2. Taxonomy of parallel programming models on hybrid platforms.

II. RELATED WORKS

The Fast Fourier Transform (FFT) plays an important role in many branches of computing applications. FFT performance is so critical that its algorithms have been

studied extensively. The Cooley-Tukey algorithm is by far the most common FFT algorithm. This algorithm factorizes the original FFT size of N into $N = N_1 \times N_2$ smaller sizes. Then it reduces the complexity from $O(n^2)$ to $O(n \log_2 n)$. Since then, many variations of the FFT have been suggested.

Many parallel FFT algorithms which based on DSP, FPGA and other different architectures are proposed in recently years. However there is so little parallel FFT algorithms based on COW with multicores. In this paper, we propose a parallel FFT algorithm which can make full use of the computing capability provided by this architecture. Linux has a well support for much parallel architecture nowadays. And because of its mature implementation of MPI and OpenMP, it is the best platform for parallel computing.

III. PARALLEL FFT ALGORITHM

Our parallel FFT algorithm can be separated into three steps. The first step is that original data is need to be rearranged in bit-reverse order and is divided into *p* blocks, where *p* is the total number of computing nodes and *N* is the size of data for FFT. The second step is the actual transformation. For a serial FFT algorithm, $\log_2 N$ steps of butterfly operations are required. During the second step where the transform is executed, these operations are decomposed into *p* nodes or *cp* CPUs, where *c* is the cores of every node. In other words, every node completes the first $\log_2(N/P)$ stages of original N-point FFT where data rearranged is not needed. The last step is the main node finishes the rest stages of iterations with the help of other nodes. Without loss of generality, *N*, *p* and *c* are all powers of 2.

Figure 3 shows the executing produce of input data (f0~f7). The number of workstation is 2. Arrow crosses stand for butterfly operation.

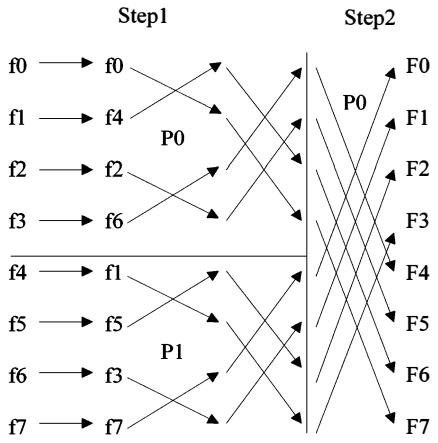

Figure 3. All butterflies for N=8.

A. *Prepare data for FFT*

In this step, the data of size *N* is bit-reversed by *c* cores of main node (see the line 3 in Algorithm 1). Then the reordered data is scattered into *p* blocks in which the data size is *N/p*.

Algorithm 1 Preprocessing (Step 1)

1: Input: **a**=$(a_0, a_1, ..., a_{N-1})$
2: Output: **b**=$(b_0, b_1, ..., b_{N-1})$
3: **#pragma omp parallel for**
for i←0 to N
4: $b_i = a_{bit\,Reverse(i)}$
5: **end for**
6: Scatter data into p nodes, every node process N/p elements and stored in d = $(d_0, d_1, ..., d_{N/p-1})$

B. *First $\log_2(N/p)$ stages butterfly*

During this produce, there is no communication between nodes because all operations adopted the received *N/p* data locally. This computation is decomposed into *c* cores. The algorithm can be separated into 2 steps. The first step is executing the first $\log_2(N/(pc))$ stages of *N/P-point* FFT where the data used by every core reside one by one(seen form the lines 4 to 16 in Alogrithm 2).Then the parallel LocalFFT algotithm represents the rest of $\log_2 c$ stages. The details are showed in algorithm2.

Algorithm 2 LocalFFT(Local N/P-Point FFT by multicores on each node)

1: Input / Output: d = $(d_0, d_1, ..., d_{n-1})$ (n=N/p)
2: ▷ Step1: The first $\log_2(N/(pc))$ stages of N/p-point FFT by c cores
3: **#pragma omp parallel num_threades(c)** /* c is the number of threads */
4: do part←n/c
5: **for** s ← 1 to $\log_2 part$
6: do m← 2^s
7: $\omega_m \leftarrow e^{2\pi i/m}$
8: **for** k ← $c_i part$ to $(c_i+1)part$ by m
/* c_i is the thread ID */
9: **for** j←0 to m/2-1
10: $a_{k+j} \leftarrow a_{k+j} + \omega_m^j a_{k+j+m/2}$
11: $a_{k+j+m/2} \leftarrow a_{k+j} - \omega_m^j a_{k+j+m/2}$
12: **end for**
13: **end for**

14: **end for**
15: **for** s←1 to $\log_2 c$
16: do m← $2^{s+\log_2 part}$
17: $\omega_m \leftarrow e^{2\pi i/m}$
18: **for** k←0 to n-1 by m
19: **#pragma omp parallel for**
20: **for** j←0 to m/2-1
21: $a_{k+j} \leftarrow a_{k+j} + \omega_m^j a_{k+j+m/2}$
22: $a_{k+j+m/2} \leftarrow a_{k+j} - \omega_m^j a_{k+j+m/2}$
23: **end for**
24: **end for**
25: **end for**

C. *The last log2p-stage*

In this method, every node sends out either of the upper or the lower *n*/2 data to the paired node. So the computation load among these nodes is well balanced. The amount of data exchange is 2*n*.

Algorithm 3 the rest stages

1: Input / Output: d = $(d_0, d_1, ..., d_{n-1})$ (n=N/p)
2: **for** s←1 to $\log_2 c$
3: do m← $2^{s+\log_2 n}$
4: **if** (rank%2 == 0) then /* even node */
5: Send upper n/2 data stored in d[n/2]-d[n-1] to the (i+1)th node.
6: Receive n/2 data from (i+1)th node and store them to upper n/2 location of d[]
7: do LocalFFT $(d_0, d_1, ..., d_{n/2-1})$
8: Send upper n/2 data stored in d[n/2]-d[n-1] to the (i-1)th node
9: Receive the n/2 data from (i+1)th node and store them back to upper n/2 location of d[]
10: **else**
11: Send lower n/2 data stored in d[0]-d[n/2-1] to the (i-1)th node
12: Receive n/2 data from (i-1)th node and store them to lower n/2 location of d[]
13: do LocalFFT $(d_0, d_1, ..., d_{n/2-1})$
14: Send lower n/2 data stored in d[0]-d[n/2] to the (i-1)th node
15: Receive the n/2 data from (i-1)th node and store them back to lower n/2 location of d[]
16: **end if**
17: **end for**

IV. RESULTS OF THE EXPERIMENT

The COW system proposed in this paper is comprised of four HPxw4400 dual-core servers connected via 1000Mbps. The software environment of the system is Red Hat Enterprise Linux AS 5, and the parallel operating environment is MPICH1.2.7, and the compiler is GCC. Each HPxw4400 dual-core servers as a computing node in the COW with CPU of Inter Pentium D 3.40GHz, 2.00 GB of memory, 80 GB of hard discs. A network adapter of Broaadcom NetXtreme Gigabit Ethernet Network Connection Construct a local area ethement by interconnecting all these computing nodes via 1000Mbps switch each node in the COW can be added or deleted according to future demand. Therefore, the COW is extensible.

The parallel FFT algorithm described in this paper was tested and the execution time for completing all stages was measured. The result is shown in Table 1.

TABLE I. TIME FOR PARALLEL FFT ALGORITHM

Data Size	Number of workstation (2)		Number of workstation (4)	
	MPI	*MPI+OpenMP*	*MPI*	*MPI+OpenMP*
2M	1.588654	1.052151	1.103251	0.832051
8M	8.062108	5.003131	5.404251	1.702432
32M	35.578878	18.009979	19.010979	6.006979

The result shows that MPI+OpenMP strategy requires much fewer time than pure MPI. As a result, the proposed algorithm using hybrid MPI+OpenMP is effective in utilizing the parallel processing capability of the architecture and achieves scalable performance.

V. CONCLUSION

This paper proposes a parallel FFT algorithm based on the MPI+OpenMP hybrid parallelization paradigm under the environment of cluster of workstation with multi-core. Hybrid programming model is better than pure MPI, and it can make full use of the cluster architecture. The results from the improved algorithm which focus on exploring the design of parallel algorithms and load balancing has shown that hybrid paradigm can achieve high efficiency and good scalable performance.

REFERENCES

[1] Julita Corbalan, Alejandro Duran, Jesus Labarta. "Dynamic Load Balancing of MPI+OpenMP applications". Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04).

[2] Series of teaching materials prepared by multi-core group. Multi-core programming . Tsinghua University Press, Beijing. 2007.

[3] Zhou WeiMing. Multi-core computing and programming [M]. Huazhong University of Science and Technology Press, Wuhan.2009.

[4] Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein(America). Introduction to Algorithms. Machinery Industry Press, Beijing. 2008.

[5] Zhang LingBo, Chi XueBin , Mo ZeRao, Li Yan. Introduction to Parallel Computing. Tsinghua University Press,Beijing.2006.

[6] Michael J.Quinn, Parallel Programming in C with MPI and OpenMP. Tsinghua University Press, Beijing. 2004.

[7] Chen GuoLiang. Practice of Parallel Algorithms, Higher Education Press, Beijing. 2004.

[8] Rolf Rabenseifner, Georg Hager, Gabriele Jost. "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes". Proceedings of the Parallel, Distributed and Network-based Processing . 2009 IEEE. DOI 10.1109/.43.

[9] Jun Ho Bahn, Jungsook Yang, Nader Bagherzadeh. "Parallel FFT Algorithms on Betwork-on-Chips". Proceedings of the fifth international conference on information technology. 2008 IEEE.DOI 10.1109/ITNG.2008.55.

[10] Jun Tan, Xingshu Chen, Long Xiao. "An Optimized Parallel FFT Algorithm on Multiprocessors with Cache Technology in Linux". Proceedings of the 2008 International Symposium on Computer Science and Computational Technology. 2008 IEEE. DOI 10.1109/ISCSCT.2008.252.