MATCH
Communications in Mathematical and in Computer Chemistry

MATCH Commun. Math. Comput. Chem. **60** (2008) 349-358

ISSN 0340 - 6253

Computing the Permanental Polynomial of C_{60} in Parallel *

Heng Liang [a] † **Hui Tong** [b] **Fengshan Bai** [a]

[a] Department of Mathematical Sciences, Tsinghua University
Beijing, 100084, P.R.CHINA.

[b] School of Science, Beijing University of Posts and Telecommunications
Beijing, 100876, P.R.CHINA.

(Received September 12, 2007)

Abstract

The permanental polynomial of buckminsterfullerene C_{60} is computed in parallel in this paper. The basic idea of the method is simply divide-and-conquer to achieve both computational precision and speed. The load balancing strategies for the permanental polynomial computation are further improved with the help of the theory of parallel machine scheduling in combinatorial optimization. The special structural properties of the matrices are considered intensively so that very high parallel efficiency is acquired. The algorithm developed in this paper can handle the permanental polynomial of C_{60} and even larger fullerenes, which are of interest in applications.

1 Introduction

The permanental polynomial of a graph is of interest in chemistry [14]. It is defined as

$$P(G, x) = per(xI - A), \tag{1}$$

where A is the adjacency matrix of a graph G with n vertices, and I is the identity matrix of order n. Here $per(A)$ denotes the permanent of the matrix A, which is defined as

$$per(A) = \sum_{\sigma \in \Lambda_n} \prod_{i=1}^{n} a_{i\sigma(i)}, \tag{2}$$

where Λ_n denotes the set of all permutations of $\{1, 2, ..., n\}$. The definition of permanent $per(A)$ looks similar to that of determinant $det(A)$. However it is much harder in computation. It is proved that

*Supported by National Science Foundation of China 10501030.

†E-mail address: hliang72@gmail.com, hliang@math.tsinghua.edu.cn

computing the permanent is a $\#P$-complete problem [13], even for matrices with only three nonzero entries in each row and column [4].

The computation of permanent and permanental polynomial recently has attracted attention in chemical graph theory [1,3,5]. Gutman and Cash suggest the following as an important open question [5]: *Can more efficient algorithms be developed so that permanental polynomial for larger systems can be studied?*

A numerical method for permanental polynomials of graphs is proposed [8]. It relates the computation of the permanental polynomial to that of permanent using Fast Fourier Transformation (FFT). All fullerenes in the set $C_{\leq 50}$ (that includes 812 isomers) are computed, and the properties of coefficients and zeros of permanental polynomials of fullerenes are analyzed based on the data obtained [12]. The algorithm works up to C_{56}. Under double-precision arithmetic, the computation of C_{56} is already borderline in precision. Parallel algorithm is proposed in this paper to improve the ability of computation of the permanental polynomials. The idea of the method is simply divide-and-conquer. It is possible to compute C_{60} and even larger fullerenes, which are of interest in real-world applications.

In the next two sections, we give a brief instruction to existing practical algorithms for permanent and permanental polynomial of sparse matrix, especially emphasizing the problems of fullerene-like structure, which are related to designing the parallel algorithm. Using the parallelized algorithm of the hybrid method is developed in section 4, the permanents and the permanental polynomials of the fullerenes are computed. In section 5, a more efficient load balancing strategy for computing the permanental polynomial is proposed with the help of the special structural properties of the problem itself and the theory of parallel machine in combinatorial optimization. Very high parallel efficiency is acquired. At last section, the computation of permanental polynomial of buckminsterfullerene C_{60} is presented.

2 Algorithms for Permanents

Algorithms for permanental polynomials are based on those for permanents. The best-known algorithm for precise evaluation of the permanent of general matrix is due to Ryser [11] and later improved by Nijenhuis and Wilf [10]. It is $O(n2^{n-1})$ in complexity. This method, which is denoted by R-NW, only works for small matrices.

It is possible to make methods for permanents faster when the special structural properties of matrices are considered. In the chemical graph theory, the first fast algorithm for permanent of fullerenes is given by Cash [1].

Taking the advantage of the structure properties extensively, a hybrid method is proposed [9]. It is

an efficient algorithm for fullerene-like matrices. Consider an expansion

$$per\begin{pmatrix} a & b & \mathbf{x}^T \\ \mathbf{y_1} & \mathbf{y_2} & \mathbf{Z} \end{pmatrix} = per\begin{pmatrix} 0 & 0 & \mathbf{x}^T \\ \mathbf{y}_1 & \mathbf{y}_2 & \mathbf{Z} \end{pmatrix} + per\begin{pmatrix} a\mathbf{y_2}+b\mathbf{y_1} & \mathbf{Z} \end{pmatrix} \tag{3}$$

$$= per(A_1) + per(A_2)$$

where a and b are scalars, $\mathbf{x}^T$ is an $(n-2)$-dimensional row vector, $\mathbf{y}_1$ and $\mathbf{y}_2$ are both $(n-1)$-dimensional column vectors, and $\mathbf{Z}$ is an $(n-1)\times(n-2)$ matrix. Combining the expansion (3) with R-NW algorithm, a hybrid algorithm is constructed.

Algorithm H_per(Hybrid) [9]

Input: A – an $n \times n$ 0-1 valued matrix .

Output: $P = H\_per(A)$

step1: Find the minimal number of nonzero entries s in one row or column of A,

step2: **if** $n > 2$ and $s < 5$, **then** divide A into A_1, A_2 as (3), and

$P = H\_per(A_1) + H\_per(A_2)$

else return by $R\text{-}NW(A)$.

3 Algorithms for Permanental Polynomials

An algorithm for computing the permanental polynomial of adjacency matrixes of chemical graphs is developed by Cash [2]. A symbolic algorithm, which relies on the computation of second order partial derivatives, is later proposed by the same author [3]. The symbolic algorithm is approximately 50 times faster for C_{40}. However the computer memory required for symbolic manipulations would grow fast with n.

A numerical method for computing permanental polynomials of fullerenes is proposed by Huo, Liang and Bai [8], which adapts the hybrid method and multi-entry expansion to FFT. It works up to C_{56} for fullerene-type graphs. The algorithm, named as H_ FFT, is given as follows.

Algorithm 2.3 H_FFT
Input: A – an $n \times n$ 0-1 valued matrix.
Output: $(a_0, a_1, \cdots, a_n)$ – the coefficients of $per(xI - A)$.

step1: Let $\omega_{n+1} = e^{(-2\pi i)/(n+1)}$ be the $(n+1)$-th root of unity,

take $x_j = \omega_{n+1}^j$ $(j = 0, 1, \cdots, n)$;

step2: Computing $p_j = per(x_jI - A)$ for $j = 0, 1, \cdots, n$;

step3: Do inverse Fourier transform for $(p_0, p_1, \cdots, p_n)$ to obtain a^{ifft};

step4: $(a_0, a_1, \cdots, a_n) = round(a^{ifft})$.

The function $round(x)$ in the algorithm gives the integer that is closest to x. Note that $per(x_jI - A)$

and $per(x_{n-j}I - A)$ are conjugate to each other. Hence only $\frac{n}{2}+1$ permanents are needed to compute in step 2. The coefficients of a permanental polynomial are all integers, while inverse Fourier transformation gives output in real numbers. The rounded part of the FFT shows the computational precision to some extent. Take

$$error = \max_{1\le k\le n+1}\{|round(a_k^{ifft}) - a_k|\}$$

be a measure of the computational precision. Table 1 gives errors of fullerenes computed.

Table 1 : The computational precision with n

Fullerene	C_{20}	$C_{30}(C_{2\nu})$	$C_{40}(C_1)$	$C_{50}(D_{5h})$
error	4.37×10^{-11}	3.44×10^{-8}	1.49×10^{-5}	1.76×10^{-3}

Fullerene	$C_{52}(D_2)$	$C_{54}(D_3)$	$C_{56}(T_d)$
error	1.47×10^{-2}	8.16×10^{-2}	1.44×10^{-1}

The errors clearly increase with n, and it would likely exceed tolerant range under the double-precision arithmetic when $n > 56$. This is reasonable, since the double precision only gives about 16 digits of accuracy, and for C_{56}, the $per(x_jI - A)$ is roughly $O(10^{15})$. So an error of 0.144 might actually be too good considering the huge amount of operations each with a rounding error in that range. In order to compute $C_n(n > 56)$, including the famous buckminsterfullerene C_{60}, more elaborate operations must be considered.

4 A Parallel Algorithm for Permanental Polynomials

It is natural to consider higher precision computation. Numerical experiments show that stability and robustness are good for C_{60} when the quadruple precision is used. However, the running time with quadruple precision is over 20 times more than that with double precision. It is so time-consuming that higher precision computation is hardly acceptable for C_n $(n > 56)$.

Divide-and-conquer is one way to make the current method to go beyond C_{56}. The value of $per(x_jI - A)$ can be divided into some smaller parts such that the 16 digits of accuracy is enough for each part. Then all the parts are added with quadruple precision. In this way the problem of precision can be overcome. The time cost of FFT is negligible compared with the exponentially growing cost of calculating the required permanents. Hence the parallelization of algorithm H_per is essential. Firstly, the $n \times n$ matrix A is divided into a series of $(n-d)\times(n-d)$ matrices by using the formula (3) repeatedly so that the permanent of each $(n-d)\times(n-d)$ matrix can be expressed exactly with the double-precision. Then the $(n-d)\times(n-d)$ matrices are computed in parallel. The d is called the depth of pre-expansion.

Based on the algorithm H_per, the following parallel method PH is constructed. Where $A_k^{(w)}$ denotes the w-th $(n-k)\times(n-k)$ matrix.

Algorithm PH (Parallel H_per)

STEP 1 Let n be the order of matrix A, num be the number of CPU's used,

$A_0^{(1)} = A$, $s = 1$, set d be the depth of pre-expansion

STEP 2 For k=1:d

t=0,

For w=1:s

divide $A_{k-1}^{(w)}$ into $A^{(1)}$ and $A^{(2)}$ as (3), $A_k^{(t+1)} = A^{(1)}$, $A_k^{(t+2)} = A^{(2)}$, $t = t+2$;

End

s=t;

End

STEP 3 send $A_d^{(w)}(w = 1, 2, \cdots, t)$ to the num CPUs in turn and compute $per(A_d^{(w)})$

by Algorithm H_per, until all $A_d^{(w)}$'s $(1 \le i \le t)$ has been computed,

STEP 4 $P = \sum_{k=1}^{t} H(A_d^{(w)})$

The main part of the parallel computation is clearly STEP 3 above and its costs dominate the whole computing time. Using the algorithm H_FFT, values $per(x_j I - A)(j = 0, 1, \cdots, n/2)$ are needed to compute in order to obtain the polynomial $per(xI - A)$, where A is the $n \times n$ adjacency matrix of fullerene C_n, $x_j = \omega_{n+1}^j$, where ω_{n+1} is the $(n+1)$-th root of unity in complex plane. For buckminsterfullerene $C_{60}(Ih)$, the number of submatrices in Algorithm PH is about 120 when $d = 8$.

All numerical experiments in this paper are carried on a 32-bit Intel Pentium III (1266 MH) with MATLAB as the programming languages. Table 2 and Table 3 give the parallel computation times for $per(x_0 I - A)$ and $per(x_1 I - A)$.

Table 2 : CPU time(Seconds) for $per(x_0 I - A)$,
where A is the adjacent matrix of buckminsterfullerene C_{60}

num	Time(sec)	Accelerated ratio	Parallel efficiency
1	79183.73	-	-
2	39780.43	1.99	0.9953
4	19956.08	3.97	0.9920
8	10176.25	7.78	0.9727
16	5326.57	14.87	0.9291
32	3134.42	25.26	0.7895

Table 3 : CPU time(Seconds) for $per(x_1I - A)$,
where A is the adjacent matrix of buckminsterfullerene C_{60}

num	Time(sec)	Accelerated ratio	Parallel efficiency
1	125528.59	-	-
2	62184.69	1.99	0.9964
4	31400.23	3.95	0.9866
8	16002.51	7.74	0.9680
16	8513.55	14.56	0.9097
32	4930.22	25.14	0.7855

The computation times for $per(x_jI - A)$ $(j > 1)$ are similar to that of $per(x_1I - A)$ because only x_0 is real and all the others are complex. Results in Table 2 and 3 show that the efficiency of the parallel method is about 80% when num, the number of CPU's, is 32, which is acceptable in parallel computation. This can be improved further by considering the special properties deeply.

5 Load Balancing Strategies

The load balancing strategies for the permanental polynomial computation can be further improved with the help of the properties of the problem itself and models of parallel machine in combinatorial optimization. Consider the following machine scheduling model first. Assume that one has a set of n jobs $J_1, \cdots, J_n$, and m identical machines $M_1, \cdots, M_m$. Each job J_j must be processed without interruption for a time $p_j > 0$ on one of the machines. Each machine can process at most one job at a time. If all jobs are ready for processing in the very beginning, it is called **offline** machine scheduling; otherwise if jobs can only be ready for processing one by one, it is called **online**. We are not going to go into full details of this field. The following example helps in understanding the basic concepts and algorithms in parallel machine scheduling.

Example 5.1. *Table 4 gives an example with $n = 9$ jobs and $m = 3$ machines. The aim is to find an order of processing, such that the overall completion time is minimized.*

Table 4: Processing Times of Jobs (n=9)

Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8	J_9
Times	*3*	*2*	*2*	*4*	*4*	*5*	*8*	*7*	*10*

The results in Figure 1–3 suggest that the order of processing is a crucial issue. Figure 1 shows the result of the order in natural, which is straightforward for online problems. Figure 2 shows the result of the non-increasing order in processing times, and Figure 3 shows the result of the optimal processing order for this example. Finding the best processing order would be hard normally. The result of Figure 2 shows that the order of non-increasing is a good strategy, which is only available for offline problems.

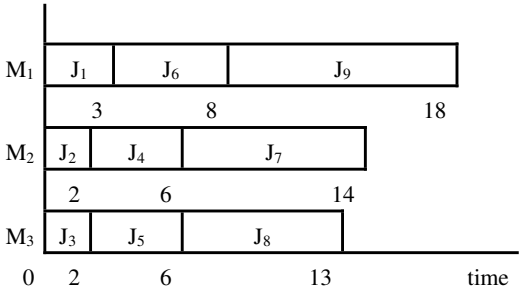

Figure 1: The Natural Order

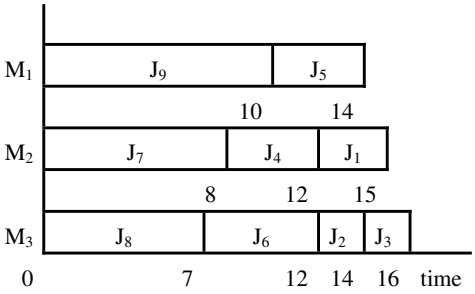

Figure 2: The Non-increasing Order in Processing Times

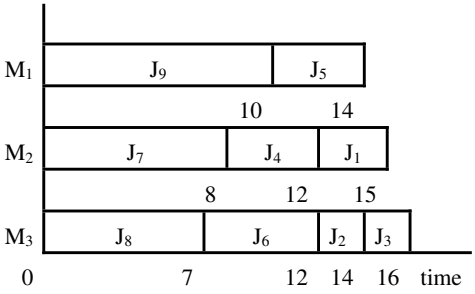

Figure 3: The Optimal Order

Algorithm LS is designed for the online parallel machine scheduling problems, where jobs are processed in its natural order of coming. Graham [6] gives the worst-case analysis of the scheduling heuristics and shows that Algorithm LS has a worst-case ratio of $2 - \frac{1}{m}$, where m is the number of machines available. If the jobs are sorted in the non-increasing order of processing times for offline problems, then there is an algorithm known as LPT. It is proved by Graham that Algorithm LPT has an improved worst-case ratio of $\frac{4}{3} - \frac{1}{3m}$ [7].

The scheduling problem in Algorithm PH presented in the last section is essentially online. One would only be able to send the sub-matrices to different processors in their natural order of expansion. However, it is important to notice that all matrices $x_jI - A$ $(j = 1, \cdots, n/2)$ have the same sparsity structure and thus have the same expansion process in the Algorithm PH. Only $x_0I - A$ is an exceptional since x_0 is the number 1 and x_j $(j = 1, 2, \cdots, n/2)$ are all complex numbers. The scheduling of the computation of submatrices generated by expanding $x_jI - A$ $(j = 2, \cdots, n/2)$ in Algorithm PH can be regarded as offline as long as $per(x_1I - A)$ is computed and all the computational times of the submatrices of $x_1I - A$ are obtained. The idea is as follows. Let the submatrices generated by expanding $x_1I - A$ be

$$C_1, C_2, \cdots, C_m,$$

and the processing times for them are

$$t_1, t_2, \cdots, t_m,$$

respectively. Sorting the processing times in the order of non-increasing, the results are

$$t_1^r, t_2^r, \cdots, t_m^r.$$

Re-order the submatrices correspondingly as their processing time

$$C_1^r, C_2^r, \cdots, C_m^r.$$

This gives the order of non-increasing in processing times for $x_1I - A$. Since matrices $x_jI - A$ $(j = 1, \cdots, n/2)$ have exactly the same structure as $x_1I - A$, the load balancing strategy for parallel computing in STEP 3 of Algorithm PH is naturally obtained.

Moreover, the submatrices of $x_jI - A$ $(j = 2, \cdots, n/2)$ can be scheduled all together in STEP 3 of Algorithm PH, which can further speed up the computation process.

6 Computational Results for Buckminsterfullerene C_{60}

The computational results of permanental polynomial of the famous buckminsterfullerene C_{60} is presented in this section. With Algorithm LPT as load balancing strategy, higher parallel efficiency is obtained as shown in Table 5. This is clear, if results in Table 3 and Table 5 are compared.

Table 5: Results of $per(x_2I - A)$ (d=8)

num	Time(sec)	Accelerated ratio	Parallel efficiency
1	125528.59	-	-
2	62771.94	1.99	0.9999
4	31430.37	3.99	0.9985
8	15748.20	7.97	0.9964
16	7959.24	15.77	0.9857
32	4100.71	30.61	0.9566

When the submatrices of $x_jI - A$ $(j = 2, \cdots, n/2)$ are scheduled all together at STEP 3 of Algorithm PH, even higher parallel efficiency is achieved. The results are given in Table 6.

Table 6: Results of $per(x_jI - A)$ $(j = 2, \cdots, 30)$ (d=8)

num	Time(sec)	Accelerated ratio	Parallel efficiency
1	3599529.54	-	-
2	1799774.60	2.00	1.0000
7	514229.54	7.00	1.0000
32	112518.48	31.99	0.9997
512	7107.02	506.48	0.9892

Finally the coefficients of permanental polynomial of buckminsterfullerene $C_{60}(Ih)$ are presented in Table 7. Numbers shown as 'coef' are the coefficients of the corresponding power indices.

Table 7: Coefficients of Permanental Polynomials of $C_{60}(Ih)$

power	coef	power	coef	power	coef
60	1	40	569449505688	20	710621056476228
59	0	39	-158182834960	19	-578519459274960
58	90	38	2380335416640	18	625754567921040
57	0	37	-804652301040	17	-500278613296800
56	3825	36	8408990831870	16	469829444630760
55	-24	35	-3389325032352	15	-359724534008944
54	102160	34	25214502300660	14	296904253790400
53	-1920	33	-11895419970480	13	-212445045923280
52	1925160	32	64398351077070	12	154600765396265
51	-72240	31	-34934337399360	11	-100775700006240
50	27245040	30	140511485229952	10	64144215349698
49	-1700880	29	-86082995358720	9	-36955969021840
48	300943940	28	262623196342200	8	20115770813385
47	-28129920	27	-178262235393200	7	-9828560060520
46	2662284480	26	421523399452800	6	4379799518800
45	-347730064	25	-310447923987216	5	-1700971266048
44	19206772020	24	582246374792420	4	572188221600
43	-3338031600	23	-454705832096640	3	-156907042080
42	114493986320	22	693042081727920	2	33654621840
41	-25522356960	21	-559816055470000	1	-4912398240
				0	395974320

References

[1] G. G. Cash, A fast computer algorithm for finding the permanent of adjacency matrices, *J. Math. Chem.* **18** (1995) 115-119.

[2] G. G. Cash, The permanental polynomial, *J. Chem. Inf. Comput. Sci.* **40** (2000) 1203-1206.

[3] G. G. Cash, A differential-operator approach to the permanental polynomial, *J. Chem. Inf. Comput. Sci.* **42** (2002) 1132-1135.

[4] P. Dagum, M. Luby, M. Mihail, U. V. Vazirani, Polytopes, permanents and graphs with large factors, *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science* (1988) 412-421.

[5] I. Gutman, G. G. Cash, Relations between the permanental and characteristic polynomials of fullerenes and benzenoid hydrocarbons, *MATCH Commun. Math. Comput. Chem.* **45** (2002) 55-70.

[6] R. L. Graham, Bounds for certain multiprocessing anomalies, *Bell System Technical Journal* **45** (1966) 1563-1581.

[7] R. L. Graham, Bounds on multiprocessing timing anomalies, *SIAM J. Appl. Math.* **17** (1969) 416-429.

[8] Y. Huo, H. Liang, F. Bai, An efficient algorithm for computing pemanental polynomials of graphs, *Comput. Phys. Commun.* **175** (2006) 196-203.

[9] H. Liang, F. Bai, A partially structure-preserving algorihm for the permanents of adjacency matrices of fullerenes, *Comput. Phys. Commun.* **163** (2004) 79-84.

[10] A. Nijenhuis, H. S. Wilf, *Combinatorial Algorithms for Computers and Calculators*, 2nd ed.; Academic Press, New York, 1978.

[11] H. Ryser, *Combinatorial Mathematics*, Mathematical Association of America, Washington DC., 1963.

[12] H. Tong, H. Liang, F. Bai, Permanental polynomials of the larger fullerenes, *MATCH Commun. Math. Comput. Chem.* **56** (2006) 141-152.

[13] L. Valliant, The complexity of computing the permanent, *Theoret. Comput. Sci.* **8** (1979) 189-201.

[14] N. Trinajstić, *Chemical Graph Theory*, 2nd ed.; CRC Press: Boca Raton, FL, 1992.