INFORMATICA, 2004, Vol. 15, No. 2, 203–218

Quick Matrix Multiplication on Clusters of Workstations

Eyas EL-QAWASMEH
Computer Science Dept., Jordan University of Science and Technology
P.O. Box 3030, Irbid 22110, Jordan
e-mail: eyas@just.edu.jo

Abdel-Elah AL-AYYOUB
Information Technology and Computing, Arab Open University
P.O. Box 1339, Amman 11953, Jordan
e-mail: ayyoub@acm.org

Nayef ABU-GHAZALEH
Department of Computer Science, State University of New York at Binghamton
Binghamton, NY 13902-6000
e-mail: nayefag@hotmail.com

Received: October 2003

Abstract. A quick matrix multiplication algorithm is presented and evaluated on a cluster of networked workstations consisting of Pentium hosts connected together by Ethernet segments. The obtained results confirm the feasibility of using networked workstations to provide fast and low cost solutions to many computationally intensive applications such as large linear algebraic systems. The paper also presents and verifies an accurate timing model to predict the performance of the proposed algorithm on arbitrary clusters of workstations. Through this model the viability of the proposed algorithm can be revealed without the extra effort that would be needed to carry out real testing.

Key words: clustered computing, matrix multiplication, MPI, parallel algorithms, performance estimation, PVM.

1. Introduction

The ever-increasing demand for high-performance computing is far beyond what sequential computers can provide. In fact, advances in natural and social sciences are mainly constrained by the limitations of computational power. The primary way of alleviating these limitations is through the advances in technology that allows higher degree of integration and faster switching circuits. Even though the technology still advances, permitting to pack more circuitry per unit of surface and decreasing transmission delays, further improvements are constrained by the speed of light. Another approach for increasing performance has been the focus in the past two decades. This approach focuses on parallel

architectures and software. Through the replication of computational elements that are interconnected in some regular structure, programs can execute on multiple hosts and access multiple memory banks. In other words, computations and memory transfers can be performed in parallel.

Parallel computers consisting of thousands of hosts are now commercially available. These high performance computers open up new frontiers in many applications since they provide solutions for numerous problems that were until today beyond the capability of conventional computing techniques. However, the benefits of parallel processing as a means to provide high computational power were limited to individuals who have access to such expensive systems. The percentage of parallel processing users compared to the computing community is rather marginal. Therefore, the early 1990's have witnessed an expanding trend of shifting from expensive proprietary supercomputers towards clusters of workstations. This transition was driven by market economies as well as the availability of commodity components for clusters of workstations.

Tremendous research efforts are being directed towards developing a novel technology that takes advantage of the increasing performance of low-cost computing systems in order to deliver high performance parallel computing. With this technology, the existing network computing infrastructure can act as one parallel computer solving one problem. This technology leverages software and hardware to reduce the effort of building high performance applications on geographically distributed computer systems.

These research efforts seek to confirm the viability of building fast, inexpensive, scalable and highly available parallel computers by using commodity workstations and personal computers. Such cost-effective "supercomputers" provide the basis for traditional parallel computing and furthermore open new horizons for novel applications such as the Internet services (Jiang, 1997).

There are two common approaches for cluster computing. The first approach exploits the idle cycles of heterogeneous shared computing resources connected via general-purpose local-area networks. This approach is efficient for loosely coupled computational models. Parallel Virtual Machine (Sunderman, 1990) and Message Passing Interface (Huss–Lederman *et al.*, 1993) are well-known examples of systems based on this approach.

In the second approach, specialized and dedicated high-speed system-area networks such as Myrinet (Boden *et al.*, 1995) and ATM (Fox *et al.*, 1987) are employed. This approach combines the rapid increase in computing performance and the recent advances in communication technology to deliver powerful parallel computers. Berkeley NOW (Agrawal *et al.*, 1995), IBM SP2 (Inktomi Corporation, 1996), and HPVM (Choi, 1998) are examples of systems based on this approach.

This paper supports the argument that network-based multicomputers are more feasible than massively parallel systems. It presents and evaluates a parallel algorithm for matrix multiplication, as a fundamental problem, on a cluster environment.

2. Related Work

Matrix multiplication is a fundamental operation that is used in many scientific areas of research such as linear algebra, signal processing, digital control, and graph theory (Agrawal *et al.*, 1994; Agrawal *et al.*, 1995; Cosnard *et al.*, 1989; Gropp *et al.*, 1999). There have been a number of approaches proposed recently for implementing matrix multiplication on distributed memory computers. The two-dimensional systolic algorithm (Chien *et al.*, 1999) and the broadcast-multiply-roll algorithm (Grayson and Geijn, 1996) are two examples. These algorithms are based on square grids with block data distribution methods.

Several attempts to implement the broadcast-multiply roll algorithm on general 2-D grids have appeared in the literature (Coppersmith and Winograd, 1990; Grayson and Geijn, 1996). Choi, Dongarra, and Walker presented a Parallel Universal Matrix Multiplication Algorithm (PUMMA) (Coppersmith and Winograd, 1990), which is a general two-dimensional grid implementation of the broadcast-multiply roll algorithm. An alternative generalization, which is also based on the two-dimensional grid, is referred to as BiMMeR (Broadcast-Multiply-Roll) (Huss–Lederman *et al.*, 1994; IBM Corporation, 1995). PUMMA and BiMMeR aimed at providing library-quality implementations of distributed matrix multiplication (Huss–Lederman *et al.*, 1993). The main difference between the two algorithms (PUMMA and BiMMeR) lies in the employed data distribution function. PUMMA uses two-dimensional block cyclic data distribution while BiMMeR uses a "virtual" two-dimensional torus wrap data distribution. Experimental results presented in (Geijn and Watts, 1996) showed that PUMMA and BiMMer algorithms achieve significant performance on the Intel Touchstone Delta (IBM Corporation, 1995).

Additional attempts on parallel matrix multiplication include a Broadcast-Broadcast Algorithm (Agrawal *et al.*, 1994), Scalable Universal Matrix Multiplication Algorithm (abbreviated SUMMA) (Gropp *et al.*, 1999), Parallel General Matrix Multiplication (abbreviated Parallel_GEMM) as a generalization of the serial GEMM routine for the Level 3 BLAS algorithm that is based on a three-dimensional data distribution (Agrawal *et al.*, 1995; Dowd *et al.*, 1995), and the Distribution-Independent Matrix Multiplication Algorithm (abbreviated DIMMA) (Choi *et al.*, 1994). The DIMMA algorithm uses a pipelined communication scheme to overlap computation and communication effectively. In addition, it uses the *least common multiple block concept* (Choi *et al.*, 1994) to obtain maximum performance.

The above-mentioned works concentrate primarily on parallelizing sequential algorithms of quadratic to cubic time complexity. In fact, most of previous works attempt parallelizing three categories of matrix multiplication algorithms; these are the standard matrix multiplication which has $O(n^3)$ time complexity, Strassen's algorithm which has $O(n^{2.8074})$ time complexity (Geijn and Watts, 1997), and the successive progress algorithms with a best case time complexity of $O(n^{2.3})$ (Cosnard *et al.*, 1989).

Recently a new algorithm for matrix multiplication with time complexity of $O(n^2)$ has been introduced as the first algorithm achieving the theoretical optimal performance (Lippert and Schilling, 1996) for certain classes of matrices (non-negative integer matrices). In this paper we present a cost effective parallel approach that further reduces the

time complexity for matrix multiplication to $O(n)$ on a cluster of n networked workstations.

The rest of the paper is organized as follows. In the next section, a non-negative integer matrix multiplication algorithm is summarized. Section 4 presents a matrix distribution function for parallelizing the non-negative integer matrix multiplication algorithm. A cluster algorithm is presented in Section 5, followed by a performance estimation model in Section 6. Next, some experimental results are presented in Section 7. Finally, the paper is concluded by a recount of obtained results.

3. Non-Negative Integer Matrix Multiplication

Matrix multiplication is a time consuming operation often encountered in fundamental problems such as linear algebra, signal processing, digital control, and graph theory. Many researchers have studied the conventional $O(n^3)$ time algorithm in an attempt to reduce its unaffordable cubic cost. Theoretically speaking, the lower bound for multiplying two $n \times n$ matrices is $O(n^2)$ since there are $2n^2$ inputs that must be examined and n^2 outputs to be computed. A general and optimal algorithm that achieves this lower bound is not yet in existence; however a special case that has been reported recently is discussed in the sequel.

Given any two non-negative integer matrices [A] and [B] of order n, their product [C]=[A]×[B] can be computed in $O(n^2)$ time using an algorithm described by Jiang and Wu (Lippert and Schilling, 1996), abbreviated JW algorithm. This algorithm improves the lowest complexity of matrix multiplication algorithm from $O(n^{2.49})$ which was given by V. Pan in 1981 to $O(n^2)$. The JW algorithm can be generalized to the field of rational numbers with a marginal increase in computing time (Lippert and Schilling, 1996). Below we describe the JW algorithm.

Algorithm JW

Compute the divider

$$x = n \times \max\{a_{ik} | 1 \leqslant i \leqslant n \text{ and } 1 \leqslant k \leqslant n\} \times \max\{b_{ik} | 1 \leqslant i \leqslant n \text{ and } 1 \leqslant k \leqslant n\} + 1.$$

Compute the pivot vector $\vec{v} = (v_1, v_2, \ldots, v_n)$ *using Horner's method as follows*:

$$v_k = b_{k1} + x\left(b_{k2} + x\left(b_{k3} + \ldots + x(b_{kn})\right)\ldots\right), \quad 1 \leqslant k \leqslant n.$$

Compute the multipliers vector $\vec{u} = (u_1, u_2, \ldots, u_n)$*, which is obtained by performing the dot product of* $\vec{v}$ *and the rows at* [A]*, as follows*:

$$u_k = \vec{v} \cdot (a_{k1}, a_{k2}, \ldots, a_{kn}), \quad 1 \leqslant k \leqslant n.$$

Compute the quotients vectors $q_{i1}, q_{i2}, \ldots, q_{i,n-2}$ *and the remainders vectors* $c_{i1}, c_{i2}, \ldots, c_{i,n}$ $(1 \leqslant i \leqslant n)$ *such that*

$$
\begin{aligned}
&u_i = q_{i1}x + c_{i1}, \quad 0 \leqslant c_{i1} < x,\\
&q_{i,j-1} = q_{ij}x + c_{ij}, \quad 0 \leqslant c_{ij} < x \quad \textit{and} \quad j = 2, \ldots, n-2,\\
&q_{i,n-2} = c_{in}x + c_{i,n-1}, \quad 0 \leqslant c_{i,n-1} < x.
\end{aligned}
$$

End JW

JW is designed for handling integer matrix multiplication. It is considered a stable algorithm since there is no rounding errors. For rational numbers, stability is not a problem for sparse matrices. However, for dense matrices, it might be a problem which needs further investigation. We should report here that there are some techniques that can be used to reduce the unstability.

4. Balanced Matrix Multiplication

In a cluster computing environment running on a loaded general-purpose network, communication latency is a crucial performance bottleneck. The matrix elements should be distributed to maintain a low communication overhead and also to guarantee balanced load on the available hosts in the cluster. In this section, we present a unified matrix distribution method that covers a wide range of matrix distribution functions and in the same time facilitate clear separation between the parallel algorithm and the choice of matrix distribution function.

Let $V = \{P_i | 1 \leqslant i \leqslant h\}$ be the set of hosts in the cluster, and let $E = \{a_{i,j} | 1 \leqslant i,j \leqslant n\} \cup \{b_{i,j} | 1 \leqslant i,j \leqslant n\} \cup \{c_{i,j} | 1 \leqslant i,j \leqslant n\}$ be the set of elements in the three $n \times n$ matrices, where [C]=[A]×[B]. The unified matrix distribution is characterized by the function $\xi = \phi \circ \rho$, where ρ: $E \to 2^E$ is the function that partitions E using a specific matrix distribution method, and ϕ: $2^E \to V$ is the function that assigns these parts to the hosts. The function ξ is a general-purpose matrix distribution that covers a wide spectrum of matrix distribution methods. As we will see later on, the above instance of ξ achieves low communication latency and also allows balanced load distribution on the available hosts.

5. Cluster Matrix Multiplication Algorithm

In his section we present a cluster matrix multiplication algorithm based on the unified matrix distribution function discussed above. In this algorithm, the host P_r, $1 \leqslant r \leqslant h$ is assigned n/h rows from [A], n/h rows from [B], and n/h rows from [C]. These $3n/h$ rows are denoted by the set $M_r = M_r^{[\mathrm{A}]} \cup M_r^{[\mathrm{B}]} \cup M_r^{[\mathrm{C}]}$. The list below formalizes the tasks performed by the JW algorithm.

– Task $\mu_r^{[\mathrm{Z}]} \equiv Find \max_r^{[\mathrm{Z}]} = \max\{M_r^{[\mathrm{Z}]}\}$, where [Z] is [A] and [B].

- Task $\pi_r \equiv Compute\ pivots\ v_k = b_{k1} + x(b_{k2} + x(b_{k3} + \ldots + x(b_{kn}))\ldots)$, $(r-1)\frac{n}{h} + 1 \leqslant k \leqslant r\frac{n}{h}$. Each host holds part of the vector$\vec{v}$, let us denotes by ${}^{r}v$ the set of elements from $\vec{v}$ assigned to and updated by P_r.
- Task $\delta_r \equiv Compute\ multipliers\ u_k = \vec{v} \cdot (a_{k1}, a_{k2}, \ldots, a_{kn})$, $(r-1)\frac{n}{h} + 1 \leqslant k \leqslant r\frac{n}{h}$.
- Task $\chi_r \equiv Compute\ final\ results\ \{c_{ij} | (r-1)\frac{n}{h} + 1 \leqslant i \leqslant r\frac{n}{h}$ *and* $1 \leqslant j \leqslant n-1\}$ *such that*

$$
\begin{aligned}
&u_i = q_{i1}x + c_{i1}, \quad 0 \leqslant c_{i1} < x,\\
&q_{i,j-1} = q_{ij}x + c_{ij}, \quad 0 \leqslant c_{ij} < x \ \textit{and}\ j = 2, \ldots, n-2,\\
&q_{i,n-2} = c_{in}x + c_{i,n-1}, \quad 0 \leqslant c_{i,n-1} < x.
\end{aligned}
$$

Using the above formulation, a parallel version of JW algorithm for non-negative integer matrix multiplication can be described as follows. The host P_r, $1 \leqslant r \leqslant h$, performs $\mu_r^{[A]}$, $\mu_r^{[B]}$ and exchanges the local extremes with all other hosts to determine the global extremes. Then, P_r performs π_r followed by a total exchange to gather up the vector $\vec{v}$ in each host's memory. Finally, P_r performs δ_r and then χ_r independently to get the final product. The parallel JW algorithm, abbreviated PJW, is outlined in Fig. 1. The algorithm makes use of a common data communication operation called *total exchange* (Rao *et al.*, 1995). A host executing a total exchange operation, denoted $\varepsilon(Ð', Ð)$, sends out the data $Ð'$ to all hosts and gathers in $Ð$ the data received from all hosts in the cluster. A straightforward implementation for the operation $\varepsilon(Ð', Ð)$ would be realized in two steps: An *asynchronous send* to all hosts to transmit $Ð'$ followed by a *block-receive* to get the portions of $Ð$ from all hosts in the cluster.

The algorithm PJW is "spawned" in the available hosts, one copy for each host, with no external coordination or synchronization. Each host gets its portion of the input, determined by the function ξ, from a local disk and writes its part of the output to a local disk. Hence, I/O operations are also concurrent in PJW.

Algorithm PJW

1. *Execute* $\mu_r^{[A]}$ *then* $\varepsilon(\max_r^{[A]}, \max_k^{[A]}$ *for* $1 \leqslant k \leqslant h$ *and* $k \neq r)$.
2. *Execute* $\mu_r^{[B]}$ *then* $\varepsilon(\max_r^{[B]}, \max_k^{[B]}$ *for* $1 \leqslant k \leqslant h$ *and* $k \neq r)$.
3. *Compute* $x = n \times \max\{\max_k^{[A]} | 1 \leqslant k \leqslant h\} \times \max\{\max_k^{[B]} | 1 \leqslant k \leqslant h\} + 1$.
4. *Execute* π_r *then* $\varepsilon({}^{r}v, {}^{k}v$ *for* $1 \leqslant k \leqslant h$ *and* $k \neq r)$.
5. *Execute* δ_r.
6. *Execute* χ_r.

End PJW

Fig. 1. Parallel JW algorithm.

An alternative formulation for PJW would be a master/slave model. The master handles communication and synchronization in the first two steps of PJW. The master spawns the slaves and all communication has to go through the master. Such a version of PJW would induce the same amount of communication and also would alleviate a little of the load on the hosts by only allowing the master to perform step 3 of PJW. However, this reduction in the hosts' computation load is insignificant (unless a very large number of hosts are involved) and it does not reduce the overall execution time of PJW. Furthermore, the PJW in Fig. 1 is superior to the master/slave version in terms of fault tolerance. A master/slave model collapses if the master goes down, which is not the case in the PJW of Fig. 1. Fault recovery measures are easily adopted by reapplying the distribution function so the new M_r's are known at the time a host goes down or comes up. We will defer discussion on PJW fault tolerance, though it is important in cluster computers built on public networks and workstations, to future research in order to stay within the boundaries of this paper's focus.

It should be noticed that the JW algorithm can also be used to multiply negative and non-negative integer matrices. This can be achieved by separating each of the factor matrices into two parts as follows:

$$[\mathrm{F}] = [\mathrm{F}]^{+} - [\mathrm{F}]^{-},$$

where $[\mathrm{F}]^{+}$ and $[\mathrm{F}]^{-}$ are obtained from [F] by taking only the positive (respectively, absolute value of negative) elements from [F] and substituting zeroes for negative (respectively, positive) elements. Thus, the product of two integer matrices [A] and [B] can be rewritten using the above formulation as follows:

$$\begin{aligned}[\mathrm{A}][\mathrm{B}] &= \left([\mathrm{A}]^{+} - [\mathrm{A}]^{-}\right)\left([\mathrm{B}]^{+} - [\mathrm{B}]^{-}\right) \\ &= [\mathrm{A}]^{+}[\mathrm{B}]^{+} - [\mathrm{A}]^{+}[\mathrm{B}]^{-} - [\mathrm{A}]^{-}[\mathrm{B}]^{+} + [\mathrm{A}]^{-}[\mathrm{B}]^{-}.\end{aligned}$$

Since $[\mathrm{A}]^{+}$, $[\mathrm{A}]^{-}$, $[\mathrm{B}]^{+}$, and $[\mathrm{B}]^{-}$ are non-negative integer matrices, the four matrix multiplications in the above expression can be carried out using the JW algorithm. Of course there will be an extra cost incurred, but this is a shortcoming of JW algorithm. PJW algorithm on the other hand makes use of the replicated processing hosts to reduce this negative influence on the execution time of the original JW algorithm.

6. Performance Results

Performance prediction models can be greatly contributes to the parallel computing (Ciegis, 2003). In this section we present a time prediction model for PJW. One of the key qualities of this model stems from the fact that machine and matrix distribution characteristics are separated from the timing parameters. The model can be used to predict the performance of PJW on arbitrary machine clusters and using arbitrary matrix distribution functions. The machine clusters can be heterogeneous; however homogenous machine characteristics give more accurate estimates.

In the sequel we denote by $ŧ(t)$ the amount of time needed to carry out the task t by one or more hosts. For instance $ŧ(\mu_r^{[A]})$ is the time taken by a single host to perform n^2/h arithmetic operations in order to find $\max_r^{[A]}$. Also, let t_{aop} denote the average time for carrying out a single arithmetic operation. With this terminology, the execution time for the algorithm PJW to multiply two matrices of order n on a cluster of h hosts is given by

$$\begin{aligned} ŧ(PJW) \;=\; & 2ŧ(\mu_r^{[A]}) + 2ŧ\big(\varepsilon(\max_r^{[A]}, \max_k^{[A]} \;\text{ for }\; 1 \leqslant k \leqslant h \;\text{ and }\; k \neq r)\big) \\ & +(2h+1)t_{aop} + ŧ(\pi_r) + ŧ\big(\varepsilon({}^r v, {}^k v \;\text{ for }\; 1 \leqslant k \leqslant h \;\text{ and }\; k \neq r)\big) \\ & +ŧ(\delta_r) + ŧ(\chi_r). \end{aligned} \tag{1}$$

The expressions in (Agrawal *et al.*, 1994) can be further refined as will be explained next. In a cluster environment, the time for a total exchange operation $\varepsilon(đ, Đ)$ can be estimated using a common model for measuring the cost of communicating a message of dimension m in a network with latency α and unit transmission cost of β, which is given by $\alpha + \beta m$ (Dongarra *et al.*, 1990). With this model the communication time in PJW can be summarized as shown below:

$$\begin{aligned} & ŧ\big(\varepsilon(\max_r^{[A]}, \max_k^{[A]} \;\text{ for }\; 1 \leqslant k \leqslant h \;\text{ and }\; k \neq r)\big) \\ & \quad = ŧ\big(\varepsilon(\max_r^{[B]}, \max_k^{[B]} \;\text{ for }\; 1 \leqslant k \leqslant h \;\text{ and }\; k \neq r)\big) = (\alpha+\beta)h, \end{aligned} \tag{2}$$

$$ŧ\big(\varepsilon({}^r v, {}^k v \;\text{ for }\; 1 \leqslant k \leqslant h \;\text{ and }\; k \neq r)\big) = (\alpha + \beta\frac{n}{h})h. \tag{3}$$

Thus, (1) can be reduced to

$$\begin{aligned} ŧ(PJW) \;=\; & 2(n^2/h)t_{aop} + 2(\alpha+\beta)h + (2h+1)t_{aop} + (2n^2/h)t_{aop} \\ & +(\alpha+\beta n/h)h + (2n^2/h)t_{aop} + \big(6n(n-1)/h\big)t_{aop}. \end{aligned} \tag{4}$$

The computation and communication characteristics of the PJW algorithm are shown in Table 1 and Table 2, respectively. The expressions in these two tables show accurate operations counts executed by each host, except for the part related to vector reduction

Table 1
Arithmetic operation counts for PJW computation tasks

Computation task	**Addition**	**Comparison**	**Multiplication**	**Division**
$\mu_r^{[A]}$	0	n^2/h	0	0
$\mu_r^{[B]}$	0	n^2/h	0	0
π_r	n^2/h	0	n^2/h	0
δ_r	n^2/h	0	n^2/h	0
χ_r	$3n(n-1)/h$	0	0	$3n(n-1)/h$
Other	1	$h-1$	$h+1$	0
Total	$(5n^2-3n)/h+1$	$2n^2/h+h-1$	$2n^2/h+h+1$	$3n(n-1)/h$

Table 2

Communication cost in PJW

Communication task	**Latency**	**Transmission Cost**
$\varepsilon(\max_r^{[\mathrm{A}]}, \max_k^{[\mathrm{A}]}$ for $1 \leqslant k \leqslant h$ and $k \neq r)$	h	h
$\varepsilon(\max_r^{[\mathrm{B}]}, \max_k^{[\mathrm{B}]}$ for $1 \leqslant k \leqslant h$ and $k \neq r)$	h	h
$\varepsilon({}^{r}v, {}^{k}v$ for $1 \leqslant k \leqslant h$ and $k \neq r)$	h	n
Total	$3h$	$2h + n$

(e.g., finding the maximum or summing up a vector) where the number of operation is considered n instead of $n - 1$. This simplifies the analysis and has negligible effect on the accuracy of the timing model.

The expressions in the above two tables indicate that the computation and communication time requirements for PJW are of order $\mathrm{O}(n^2/h)$ and $\mathrm{O}(n + h)$, respectively. From these complexities, we deduce that by increasing the number of hosts we can end up in a faster PJW. However, involving larger numbers of hosts means higher communication complexity, and hence it is possible to end up in a slower PJW. The next section investigates this point.

7. Experimental Results

In this section, the performance of the PJW algorithm is investigated on a cluster of 32 Pentium III workstations connected together via 100Mb/s Ethernet segments. The workstations are running under Linux with PVM and MPI in support of cluster computing.

PVM is a message passing system that enables a network of heterogeneous computers to be used as a single distributed memory parallel computer. PVM can be used at several levels. "At the highest level, the *transparent* mode, tasks are automatically executed on the most appropriate host. In the *architecture-dependent* mode, the user specifies the types of hosts suitable to run a specific task. In the *low-level* mode, the user may specify a particular host to execute a particilar task" (Sunderman, 1990). In all of these modes, PVM takes care of necessary data conversions from one host to another as well as low-level communication details (Huss–Lederman it et al., 1993; Sunderman, 1990).

MPI, on the other hand, is a software for message passing, proposed as a standard by a broad committee of vendors, implementers, and users (Huss–Lederman *et al.*, 1993). MPI is portable and flexible software that is used widely. It is considered as a standard for writing message-passing programs in which higher level routines and abstractions are built upon lower level message passing routines.

In our experiments we implemented PJW using both PVM and MPI. These two implementations of the PJW algorithm were tested using matrices of orders ranging from 1,000 up to 10,000 with strides of 500. In order to avoid overflow exceptions for large matrix orders, small-valued non negative matrix elements were used. The experiments

have been repeated using 4, 8, 16 and 32 hosts for both implementations with a total of 152 test runs.

The employed cluster environment is a typical academic installation with dedicated local area network. In order to avoid pointless scenarios, the experiments were conducted after midnights where no unknown loads on the hosts. Furthermore, unnecessary time stealing daemons were blocked at the time of experimentation.

Fig. 2 shows the real execution times for PVM-based PJW on 4, 8, 16, and 32 hosts and for matrix orders ranging from 1,000 to 10,000 elements. Although the algorithm runs

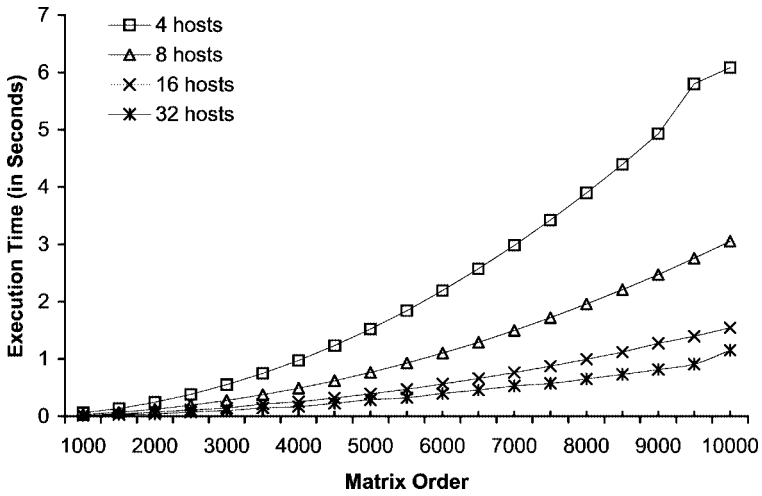

Fig. 2. The real execution times for PVM based PJW.

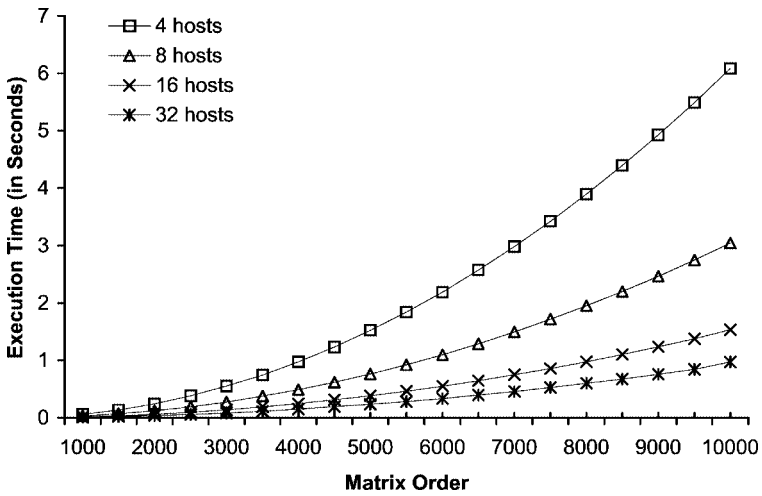

Fig. 3. The real execution times for MPI based PJW.

faster on a larger number of hosts, the gain in the speedup factor is slower. For instance, the difference in execution time between 16 and 32 hosts is smaller than the difference between 8 and 16 hosts. This is due to the dominance of increased communication cost over the reduced in computation cost.

The MPI experimentation results are shown in Fig. 3. The results are somewhat similar to those for the PVM with some reduction of the execution time in favor of MPI. This reduction becomes more apparent for a larger number of hosts. This leads to the

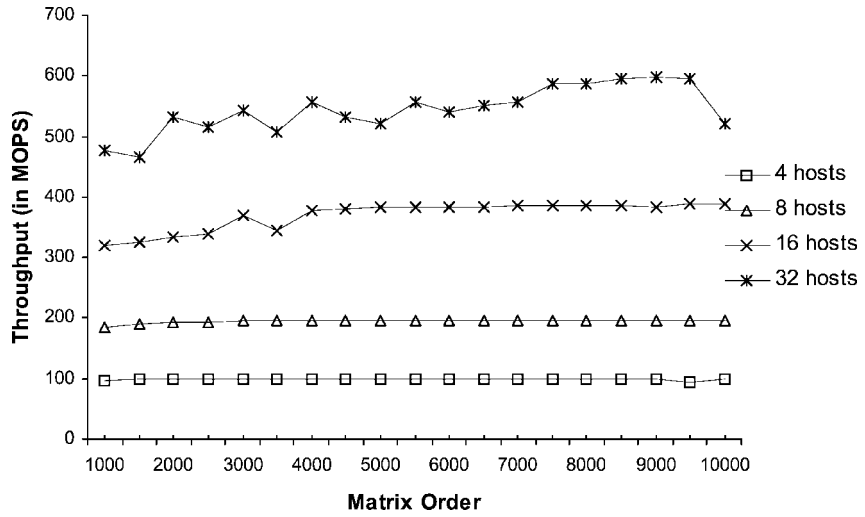

Fig. 4. The attained throughput for PVM based PJW.

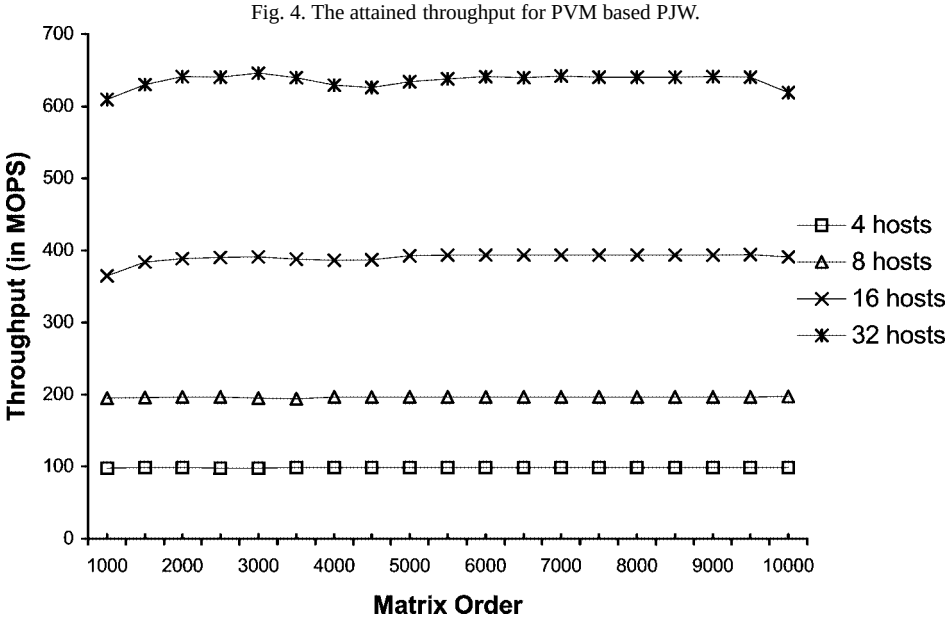

Fig. 5. The attained throughput for MPI based PJW.

observation that MPI performs faster message passing than PVM in this experiment.

Figs. 4 and 5 show the attained throughput (in millions of arithmetic operations per second) for PJW using PVM and MPI, respectively. These throughputs compare favorably to today's most expensive supercomputers in terms of cost/performance ratio. Hence, these results confirm the viability of cluster computing as a cost-effective alternative to expensive supercomputing. The fluctuating throughput for larger numbers of hosts is due to the non-deterministic workload on the public network. Although the experiments were

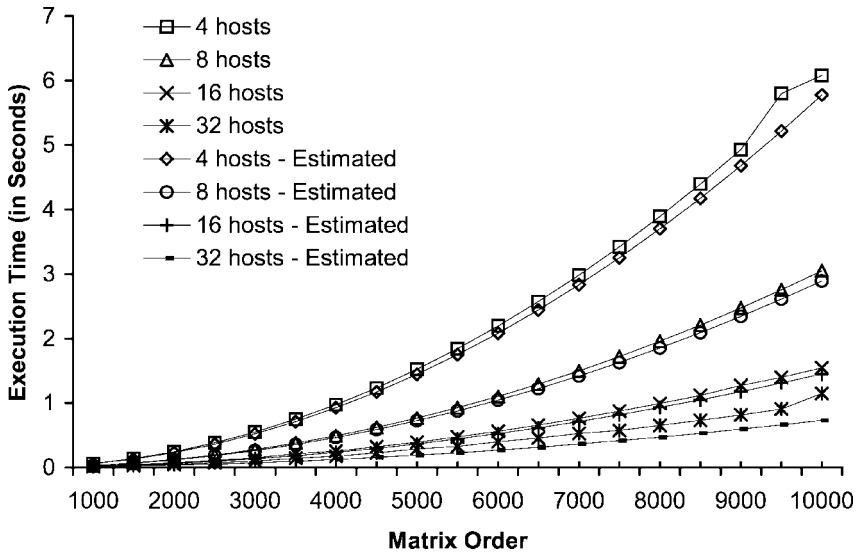

Fig. 6. The estimated and the real execution times for PVM based PJW.

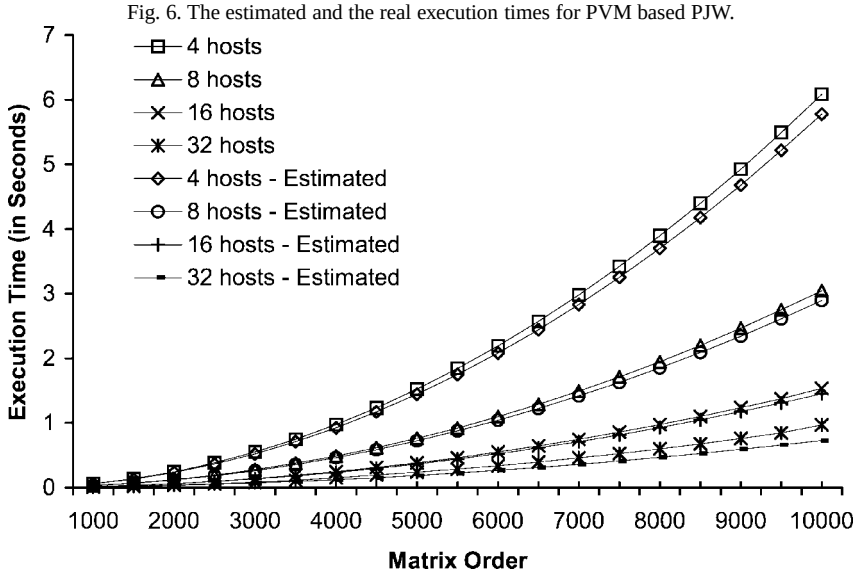

Fig. 7. The estimated and the real execution times for MPI based PJW.

conducted after midnight, one should expect occasional traffic on the network.

The final experiment was conducted to verify the correctness of the proposed timing models for PJW. The expressions in Tables 1 and 2 are plotted in Figs. 6 and 7 using PVM and PMI time parameters, respectively. Arithmetic execution times are the same for both cases, however the communication latency has been empirically obtained by measuring the time needed to transmit a 4-byte message between a pair of hosts. The difference between the recorded time and the ideal time (for a known network bandwidth) represents the communication latency. To obtain an accurate estimate, the measured communication latency has been averaged over one hundred tests. The network latency α and unit transmission cost $beta$ have been measured using PVM and then using MPI primitives. The two values pairs have record and used in the subsequent experiments. A similar number of tests have been carried out to estimate the arithmetic execution times (addition, comparison, multiplication, and division) on the employed Pentium workstations.

As can be seen from Figs. 6 and 7, the estimated execution times are quite close to the real ones. This verifies that the proposed timing model for PJW is fairly accurate, and hence it provides a means to test the viability of PJW on any cluster without taking the burden of real testing.

The key qualities that have been the driving force in designing the PJW are: linear time complexity (with sufficiently large number of hosts), low cost as it utilizes off-the-shelf networked workstations with huge aggregate memory space, and low communication overhead. One problem that still persists in the original JW algorithm is the overflow scenarios inherited from the way pivot elements are computed. PJW algorithm is not meant to solve the overflow problem of JW, which is deferred to future research. One way to solve this problem is by using emulated large integers. Of course this adds some overhead to JW and PJW execution times, however the gained improvement in time complexity of the PJW worth paying this extra overhead.

8. Concluding Remarks

The JW matrix multiplication algorithm was the first that reaches the optimum of O(n^2) to multiply two integer matrices of order n (Lippert and Schilling, 1996). In this paper we proposed and evaluated a parallel JW algorithm (abbreviated PJW) that reduces the complexity to O(n^2/h), where h is the number of hosts in the cluster. The algorithm has been tested on a cluster of 32 Pentium hosts connected together using 100 mps Ethernet segments. The obtained results show little timing difference in favor of MPI over PVM, yet confirm the viability of cluster computers as a cost-effective alternative for expensive supercomputers. The 32-host cluster attained over 650 MOPS, which is comparable to modern supercomputer performance. Another key ability of cluster computing is the huge aggregate memory space that would be unachievable in a single system. The matrix order 10,000 requires 1.2 GB of real memory to store the three matrices in JW or PJW. High memory requirement would be hardly supported in a multi-user supercomputer system for some problems, hence making cluster computers with large aggregate memory spaces a suitable host for applications demanding large computational power.

References

Agrawal, R., F. Gustavson and M. Zubair (1994). A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM Journal of Research and Development*, **38**(6), 673–681.

Agrawal, R., S. Balle, F. Gustavson, M. Joshi and P. Palkar (1995). Three dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, **39**(5), 575–583.

Anderson, T., D. Culler, D. Patterson and the NOW Team (1995). A Case for NOW (Networks of Workstations). *IEEE Micro*, **15**(1), 54–64.

Boden, N., D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic and W. Su (1995). Myrinet – a gigabit-per-second local-area network. *IEEE Micro*, **15**(1), 29–36.

Chien, A., M. Lauria, R. Pennington, M. Showerman, G. Iannello, M. Buchanan, K. Connelly, L. Giannini, G. Koenig, S. Krishnamurthy, Q. Liu, S. Pakin and G. Sampemane (1999). Design and evaluation of an HPVM-based Windows NT supercomputer. *International Journal of High-Performance Computing Applications*, **13**(3), 201–219.

Choi, J. (1998). A new parallel matrix multiplication algorithm on distributed memory concurrent computers. *Concurrency: Practice and Experience*, **10**(8), 655–670.

Choi, J., J. Dongarra and D. Walker (1994). PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. *Concurrency: Practice and Experience*, **6**(7), 543–570.

Ciegis, R., and V. Starikovicius (2003). Realistic performance prediction tool for the parallel block LU factorization algorithm. *Informatica*, **14**(3), 167–180.

Coppersmith, D., and S. Winograd (1990). Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, **9**(3), 251–280.

Cosnard, M., Y. Robert and B. Tourancheau (1989). Evaluating speedups on distributed memory architectures. *Parallel Computing*, **10**(2), 247–53.

Dongarra, J., I. Duff, J. Croz and S. Hammarling (1990). A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, **16**(1), 1–17.

Dowd, P., S. Srinidhi, E. Blade and R. Claus (1995). Issues in the ATM support of high performance geographically distributed computing. In *First International Workshop on High Speed Network Computing*. pp. 19–28.

Fox, G., S. Otto and A. Hey (1987). Matrix algorithm on a hypercube I: matrix multiplication. *Parallel Computing*, **4**(1), 17–31.

Grayson, B., and R. Geijn (1996). A high performance parallel strassen implementation. *Parallel Processing Letters*, **6**(1), 3–12.

Geijn, R., and J. Watts (1997). SUMMA: Scalable Universal Matrix Multiplication Algorithm. *Concurrency: Practice and Experience*, **9**(4), 255–274.

Gropp, W., E. Lusk and A. Skjellum (1999). *Using MPI*, 2nd Edition. MIT Press.

Huss–Lederman, S., E. Jacobson and A. Tsao (1993). Comparison of scalable parallel matrix libraries. In *Proceedings of the Scalable Parallel Libraries Conference*, Starksville, MS. pp. 142–149.

Huss–Lederman, S., E. Jacobson, A. Tsao and G. Zhang (1994). Matrix multiplication on the Intel Touchstone Delta. *Concurrency: Practice and Experience*, **6**(7), 571–594.

IBM Corporation (1995). *Scalable POWERparallel System*.

Inktomi Corporation (1996). The Inktomi technology Behind Hotbot, *A White Paper*. Also available from `http://www.inktomi.com/Tech/CoupClustWhitePap.html`.

Jiang, C., and Z. Wu (1997). Two new algorithms for matrix multiplication and vector convolution. *International Journal of Computer Mathematics*, **63**(1–2), 23–36.

Lippert, T., and K. Schilling (1996). Hyper-systolic matrix multiplication. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Sunnyvale, California, USA. pp. 919–930.

Rao, S., T. Suel, T. Tsantilas and M. Goudreau (1995). Efficient communication using total-exchange. *Proceedings of the* 9*th International Parallel Processing Symposium*, Santa Barbara, CA. pp. 544–550.

Sunderman, V. (1990). PVM: a framework for parallel and distributed computing. *Concurrency: Practice and Experience*, **2**(4), 315–339.

E. El-Qawasmeh received his BSc degree in computer science in 1985 from Yarmouk University, Jordan. He then joined the Yarmouk University as teaching assistant in the Computer Science Department. In 1992, he joined the George Washington University, Washington, D.C., USA where he obtained his MS and PhD degrees in software and systems in 1994 and 1997, respectively. In 2001, he joined George Washington University, USA as visiting researcher through a Fulbright Commission grant. He received 2001 Hijjawi research prize for computer science (best computer research for 2001 in Jordan and Palestine). His areas of interest include multimedia databases, information retrieval, and object-oriented. Dr. El-Qawasmeh is currently an assistant professor in the Department of the Computer Science and Information Systems at Jordan University of Science and technology, Jordan. He is a member of the editorial board of *Digital Information Management Journal*. He is a member of the ACM and IEEE.

A.-E. Al-Ayyoub is an associate professor of computer science at the Arab Open University. He received his BSc degree in computer science in 1986 from Yarmouk University, Jordan. He then joined the Middle East Technical University, Turkey, where he obtained his MS and PhD degrees in computer engineering in 1987 and 1992, respectively. His areas of interest include interconnection networks, parallelizing compilers, design of parallel algorithms, mobile computing, and artificial intelligence. Dr. Al-Ayyoub is an IEEE senior member. He received more than quarter a million US dollars in research grants, won two major prizes in computer science (the State Prize and Abdul-Hameed Shoman Prize), and published over 50 papers in well-known journals and conference proceedings. Dr. Al-Ayyoub supervised PhD and master students and developed several graduate and undergraduate programs in various fields of information technology. Before joining the Arab Open University, Dr. Al-Ayyoub severed in the University of Bahrain, Sultan Qaboos University – Oman, The University of Akron – Ohio, and Jordan University of Science and Technology – Jordan. His experience in teaching extends to 14 years.

N. Abu-Ghazaleh is pursuing his PhD at State University of New York at Binghamton. He got his BSc from Jordan University of Science and Technology in 2002. He then joined the state Univesity of New York at Binghamton where he got his master in 2004.

Greitasis matricų dauginimas naudojant vartotojų darbo vietų klasterius

Eyas El-QAWASMEH, Abdel-Elah AL-AYYOUB, Nayef ABU-GHAZALEH

Straipsnyje pateiktas greitojo matricų dauginimo algoritmas, skirtas vykdyti tinkle sujungtuose vartotojų darbo vietų klasteriuose. Gauti rezultatai patvirtina, kad algoritmas tinka greitam ir pigiam daug skaičiavimų reikalaujančių uždavinių, pavyzdžiui, didelės apimties tiesinės algebras uždavinių, sprendimui. Taip pat straipsnyje pateiktas skaičiavimo laiko poreikių prognozės modelis.