Parallel Computing 2 (1985) 109-118
North-Holland

Task granularity studies on a many-processor CRAY X-MP

D.A. CALAHAN

Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 4809, USA

Received April 1985

Abstract. A hybrid granularity model is proposed for general concurrent solution. It is applied to the triangular factorization of a dense matrix ranging in size from 4 to 1024. Concurrency is achieved at two levels: (1) with small (micro) task granularity and (2) with large (blocked) task granularity. Relevance to a many-processor CRAY X-MP is demonstrated by simulation.

Keywords. Parallel algorithms, task granularity, microtasking, CRAY X-MP, triangular factorization

1. Introduction

1.1. Parallel architecture classifications

Parallel (concurrent) scientific architectures proposed to achieve GIGAFLOP performance tend to have one of two attributes.

1) Low Parallelism ($\leqslant 64$ processors). In this evolutionary architecture, individual pipelined vector processors with peak performance in the range of 100-500 MFLOPS are interconnected principally through a main memory. The CRAY X-MP is currently a 2-processor example [8]. These will be termed vector multiprocessors (VMPs).

2) Massive parallelism. A number of revolutionary architectures with individual processors in the range of 5-10 MFLOPS and specialized interprocessor connections have been proposed in recent years [9,10].

The same multiprocessor architecture and algorithmic attributes that have been researched for massively parallel machines—i.e., interprocessor signalling and data communication, task processing, and algorithmic partitioning—can be studied for VMP's. An advantage of such a study is that results can be compared with actual machine performance on the CRAY X-MP for $p = 2$; extrapolations to a many-processor configuration have potential near-term value for extensions of the CRAY family.

In this paper, the former class is studied by instruction level simulation of a many-processor extension of the CRAY X-MP. The intent is to give detailed insight into the algorithmic and interprocessor communication issues peculiar to VMP's.

1.2. Hybrid granularity model

At its highest level, the concurrent algorithm organization for a VMP involves the tasking concept.

0167-8191/85/$3.30 © 1985, Elsevier Science Publishers B.V. (North-Holland)

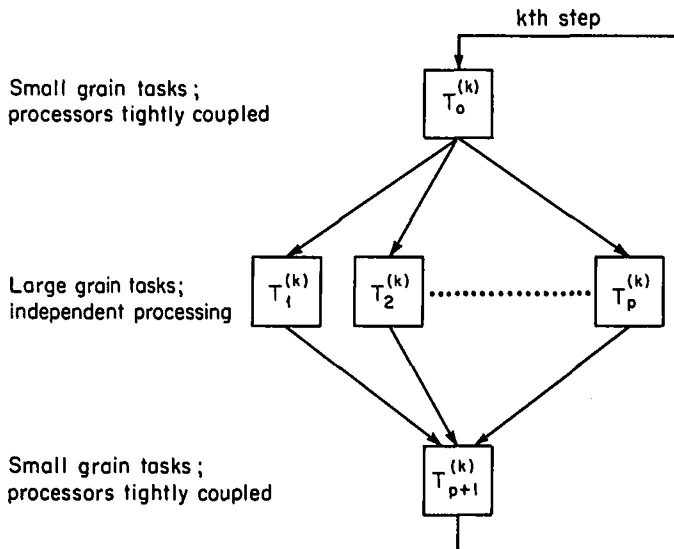

Fig. 1. Hybrid granularity computational model.

Define a *task* as a set of instructions which communicate with other tasks only at task initialization and termination. Task *granularity* (size) is the single most critical issue in classifying concurrent architectures and algorithms. An architecture and associated system software that can effectively support a variety of small tasks can certainly process large tasks effectively. Conversely, algorithms involving large independent tasks that have low inter-processor signalling or data flow per operation are the most likely to be usuable on a spectrum of concurrent architectures.

Although certain codes naturally decompose into large-grain independent tasks, most at best involve a combination of large-grain and small-grain tasking. One model natural to some physical problems is shown in Fig. 1. Here, a large-grain decoupling task $T_0^{(k)}$ is performed by all processors cooperating at the small-grain level, on the kth step of a process; this could be a field-related calculation in a physics problem [4], or, as in this paper, a block LU factorization in solution of a set of equations. This step enables p simultaneous large-grain tasks $T_1^{(k)},\ldots,T_p^{(k)}$ to be performed. A large-grain coupling task $T_{p+1}^{(k)}$, again composed of small-grain tasks, may be present, as in divide-and-conquer algorithms; alternatively, $T_{p+1}^{(k)}$ may be viewed as $T_0^{(k+1)}$ of the next iteration. This will be termed the *hybrid granularity* model. Although the concurrent

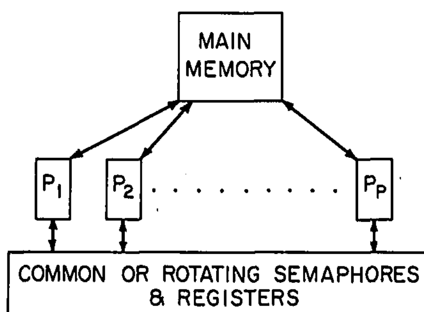

Fig. 2. CRAY X-MP extension to p processors.

tasks $T_1^{(k)} \ldots T_p^{(k)}$ usually together involve the majority of the computational workload, the time to perform $T_0^{(k)}$ and/or $T_{p+1}^{(k)}$ may dominate the time of the concurrent tasks. The importance of efficient implementation of the small-grain steps—and of the tasking hardware they require—is open to question. In this paper, this issue is investigated for a 16-processor CRAY X-MP extension solving a common linear algebra problem [1,7].

2. Tasked equation solution

2.1. Introduction

The problem chosen for study in this paper is the triangular factorization of a matrix. Among other attributes, factorization (a) has a sequential nature so that successive interdependent tasks must be defined, and (b) permits the task size to be varied by algorithmic means so that the size can be forced to be sufficiently small to stress the tasking capability of the architecture being studied [6].

Consider the solution of the matrix equation

$$AX = B$$

for X, where A is an $n \times n$ full matrix, and X and B are vectors. A study of the solution is to be made on a p-processor X-MP, where each processor communicates with other processors through semaphores, shared registers, and main memory (Fig. 2). No privoting is involved in this model.

Two classes of tasking are to be studied.

a) Large-grain tasked solution. Here the matrix is blocked and block-level operations are controlled as tasks. This blocking has been shown very important to efficient solution on the CRAY-1 [2,3] since it (1) migrates loop control overhead to the outer loops, and (2) reduces traffic between main memory and the vector register cache. Tasking overhead also decreases as the block size increases.

b) Small-grain tasked solution (microtasking [1]). Although the CRAY X-MP permits high speed semaphore and limited scalar communication between processors, vectors must be passed through main memory. There is a question of whether this combination is adequate to support that is shared and synchronized at a low level, i.e., with tight processor coupling. The above blocked solution synchronization is therefore moved down two levels by reverting to column-by-column triangular factorization. Tasking control and other overhead is now of major concern.

These solution will eventually be combined in a two-level factorization algorithm for large matrices.

2.2. Blocked factorization

2.2.1. Introduction. Let the block-partitioned LU factorization of a matrix be represented in the form

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1q} \\ A_{21} & A_{22} & & \cdot \\ \ddots & \ddots & & \ddots \\ A_{q1} & \cdots & & A_{qq} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & & \cdot \\ \ddots & \ddots & & \ddots \\ L_{q1} & \cdots & & L_{qq} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & \cdots & U_{1q} \\ 0 & U_{22} & & \cdot \\ \ddots & \ddots & & \ddots \\ 0 & \cdots & & U_{qq} \end{bmatrix} \quad (1)$$

where A_{rr}, L_{rr}, and U_{rr} are $n_r \times n_r$ matrices, and L_{rr} and U_{rr} are lower and upper triangular matrices, respectively.

We represent the triangular LU factorization of A by a series of blocked eliminated steps proceeding from A_{11} to A_{qq}, and using, at the rth step in a Crout-like reduction, the outer product of the rth row and column of blocks to partially reduce the remaining southeast corner of the matrix.

This step is notationally simplified at the rth step by collecting the partially-reduced block-row to the right of the diagonal into $A_{12}^{(r)}$ and the partially-reduced block-column below the diagonal into $A_{21}^{(r)}$; the partially-reduced diagonal block is denoted $A_{11}^{(r)}$ and the remaining southeast corner collection of blocks is $A_{22}^{(r)}$. The simplified partially-reduced matrix is

$$\begin{matrix} A_{11}^{(r)} & A_{12}^{(r)} \\ A_{21}^{(r)} & A_{22}^{(r)} \end{matrix} \tag{2}$$

where $A_{11}^{(r)}$ and $A_{21}^{(r)}$ are $n_r \times n_r$ and $n_r \times m_r$ matrices, respectively. The reduction of the next n_r rows and columns is completed as follows.

Step (1). Factor $A_{11}^{(r)}$ into lower and upper triangular form

$$A_{11}^{(r)} \leftarrow L_{rr} U_{rr}. \tag{3}$$

Step (2). Substitute into $A_{12}^{(r)}$ and $A_{21}^{(r)}$.

$$A_{12}^{(r)} \leftarrow L_{rr}^{-1} A_{12}^{(r)}, \tag{4}$$

$$A_{21}^{(r)} \leftarrow A_{21}^{(r)} U_{rr}^{-1} \tag{5}$$

which completes the formation of a block-row of U and a block-column of L. Eq. (5) requires more computation than Eq. (4) if U_{rr} has non-unitary diagonals.

Step (3). Accumulate the outer product into $A_{22}^{(r)}$.

$$A_{22}^{(r)} \leftarrow A_{22}^{(r)} - A_{21}^{(r)} A_{12}^{(r)}. \tag{6}$$

This blocked solution is strictly conservative of operation count vis-a-vis Gauss elimination [5].

2.2.2. Blocked parallelization. The accumulation step of Eq. (6) usually involves the larger number of floating point operations and so is of first concern. With p processors, Step (3) can be partitioned into

$$\left[A_{22,1}^{(r)} A_{22,2}^{(r)} \cdots A_{22,p}^{(r)}\right] \leftarrow \left[A_{22,1}^{(r)} A_{22,2}^{(r)} \cdots A_{2,p}^{(r)}\right] - A_{21}^{(r)} \left[A_{12,1}^{(r)} A_{12,2}^{(r)} \cdots A_{12,p}^{(r)}\right] \tag{7}$$

where $A_{ij,k}^{(r)}$ is a m_r/p-column partition of $A_{ij}^{(r)}$. This partition preserves the average vector length $\bar{l} = m_r$, the number of rows of $A_{21}^{(r)}$; the vector loop is executed n_r times, the row dimension of $A_{12,k}^{(r)}$. A full concurrency of p is achieved provided m_r/p is an integer. Note that this partitioning across the processors is unrelated to original blocking, except for the dimension n_r.

The substitutions of Eqs. (4) and (5) are similarly partitioned into

$$\left[A_{12,1}^{(r)} \cdots A_{12,p/2}^{(r)}\right] \leftarrow L_{rr}^{-1} \left[A_{12,1}^{(r)} \cdots A_{12,p/2}^{(r)}\right], \tag{8}$$

$$\begin{bmatrix} A_{21,1}^{(r)} \\ \vdots \\ A_{21,p/2}^{(r)} \end{bmatrix} \leftarrow \begin{bmatrix} A_{21,1}^{(r)} \\ \vdots \\ A_{21,p/2}^{(r)} \end{bmatrix} U_{rr}^{-1} \tag{9}$$

where it is assumed that $p/2$ processors are assigned to each substitution.

Eqs. (8) and (9) can be implemented in two ways. *Simultaneous* substitution into all rows of $A_{21,i}^{(r)}$ and into all the columns of $A_{12,i}^{(r)}$ yields $\bar{l} = 2m_r/p$; *conventional* substitution results in $\bar{l} = n_r/2$. The latter will be used in this experimental study, since $\bar{l}$ is independent of p.

2.3. *Microtasking* [1]

2.3.1. Factoring the diagonal blocks. A virtue of the blocked algorithm is that, once the diagonal block factorization of Step (1) is completed, the multiplication and substitution steps are readily partitioned as above, and involve large, concurrent similarly-sized tasks with $\bar{l} = n_r$ or $\bar{l} = n_r/2$. Any inefficiency due to interprocessor communication will therefore occur in Step (1).

The following study concerns the application of p tightly coupled processors to this block factorization; it is a special case of the overall factorization problem when $n_r \leqslant 64$, the maximum length of a vector on the CRAY X-MP.

2.3.2. Algorithm. let $V_{j:k,i}$ represent a partial ith column of the matrix, beginning at row j and ending at row k, and let s_{ij} be and (i, j) scalar element of the matrix. Then Steps (1)–(2) are replaced by the following to completely reduce the rth column.

Step (1). Reciprocate pivot.

$$s_{rr} \leftarrow 1/s_{rr}. \tag{10}$$

Step (2). Substitute into the rth column in $V_{1:n,r}$.

$$V_{r+1:n,r} \leftarrow s_{rr} V_{r+1:n,r}. \tag{11}$$

Step (3). Accumulate into $V_{2:n,r}$. For $i = 1,2,\ldots n-1$,

$$V_{i+1:n,r} \leftarrow V_{i+1:n,r} - s_{i,r} V_{i+1:n,i}, \tag{12}$$

$$s_{i+1,r} \leftarrow V_{i+1:i+1,r} \tag{13}$$

The vector inner loop requires only one load and no store per add multiply in (12). However, Eq. (13) requires a wait in the inner loop until Eq. (12) is completed; this slows the inner loop performance below that of a matrix multiply [2,3].

It is proposed that a multiprocessor version involve the *simultaneous* accumulation of columns $rp-p+1 \ldots rp$ at the rth step. The ith processor is responsible for the reduction of the $rp-p+i$ column by the accumulation of previous columns. The critical phase of this process occurs in the accumulation step of (13) when a processor requires columns $V_{i+1:n,pr}$. A potential wait could occur when, for $pr-p+1 \leqslant pr-1$, $V_{i+1:n,i}$ is an operand in Eq. (12). At most, $p-1$ potential waits occur at each step, so that proportionately the greatest potential disruption occurs when r is small.

In summary, interprocessor communication occurs at the level above the vector inner loop, the lowest level consistent with vector processing and two levels below the previous blocked solution.

3. Implemention and performance evaluation

3.1. *The CRAY X-MP simulator*

A simulator that performs instruction-level timings and numerical calculation from assembly language codes has been developed for a many-processor CRAY X-MP. The simulator incorporates the semaphore, shared register, and bank conflict protocol of the 2-processor X-MP (X-MP-2), extended up to 16 processors and 256 memory banks. General instruction timing accuracy vis-a-vis the X-MP-2 is within 0.2% using a library code in one processor and an idle second processor.

[1] Since the first published use of 'microtasking' in [7], Cray Research has adapted this term to describe a library routine for small-grain tasking [13].

Rather than parameterize the results as a function of the number of memory banks, conflict checking was disabled during the simulation to be reported. Extensive simulation studies [11] have shown that, if the ratio

$$R_{bp} = \frac{\#\text{of memory banks}}{\#\text{of processors}}$$

is maintained at 16, the delay in execution timing is in the range 3–5% for a variety of codes. It has been found that this percentage affects the absolute timings uniformly, and relative timings are unaffected by ignoring conflicts. Also, the cost of simulation increases by a factor between 5 : 1 and 8 : 1 by including conflict checking.

3.2. Microtasked solutions results

3.2.1. Implementation. To illustrate the effect of coding on MP performance, two implementations of the microtasked factorization will be compared.

(1) *Code* #1. "Standard" assembly language (CAL) coding from [2] was used, closely following the previous description. Every fetch was preceded by an address test through the rotating shared registers to determine whether the operand vector had been calculated.

(2) *Code* #2. The inner accumulation loop can be written so that *pairs* of rows are reduced by a single vector operand fetch; this reduces memory traffic, address testing, and permits better floating point pipeline utilization [3] for short vectors. For example, Table 1 shows that a speedup of up to 1.47 is achieved simply by this coding improvement on a uniprocessor.

Table 1
Comparison of microtasked solution methods; timings are simulated

Matrix size (n_r)	Code #1			Code #2		
	Clocks	η	MFLOPS	Clocks	η	MFLOPS
1 processor						
4	621	1.0	6.45	593	1.0	6.75
8	1789	1.0	18.6	1441	1.0	23.1
16	6749	1.0	40.8	4618	1.0	59.6
32	30774	1.0	73.1	20097	1.0	112.
64	164214	1.0	111.0	118689	1.0	153.
2 processors						
4	524	0.592	7.64	562	0.527	7.12
8	1150	0.779	28.9	1161	0.621	28.6
16	3730	0.904	73.8	2932	0.787	93.8
32	16011	0.961	140.0	11081	0.907	203.
64	83588	0.981	218.0	61610	0.963	295.
4 processors						
8	1027	0.435	32.4	1119	0.322	29.7
16	2514	0.671	109.0	2530	0.456	109.
32	8838	0.870	254.0	7522	0.668	299.
64	43414	0.943	419.0	34464	0.861	527.
8 processors						
16	2151	0.392	128.0	2323	0.248	118.
32	5660	0.679	398.0	6577	0.382	342.
64	23946	0.855	760.0	23612	0.502	770.
16 processors						
32	4993	0.385	451.0	5929	0.212	379.
64	15339	0.667	1186.0	20350	0.364	893.

3.2.2. Evaluation. Define an efficiency as

$$\eta = \frac{\text{uniprocessor time}}{(\text{multiprocessor time}) * p}$$

To achieve $\eta = 1$, it is necessary (a) for the computation to be evenly divided among processors, (b) for the task control (signalling) time to be zero and (c) for the operand wait time to be zero. Here, (a) and (c) are largely algorithm dependent, while (b) depends more on the coding.

Table 1 shows, for $p = 2$, an efficiency of 0.98 is achieved for matrices of the largest size considered ($n_r = 64$). This near-optimal performance implies that (a) operand wait time (noted above) is minimal, and (b) the address test associated with every accumulation is overlapped by other computation; indeed, simulation shows that long-vector operations intrinsic to the accumulation process *completely overlap* (and thus mask) *task control operation*. This possibility is peculiar to vector processors. As p increases, the likelihood of operand-waits increases, and η decreases to 0.667 for $p = 16$.

The uniprocessor speed advantage of code #2 is observed to vanish as p increases (Table 1). Each processor now handles the reduction of two adjacent rows, a larger task than in code #1. By so dividing the sequential solution process into fewer but larger tasks, the likelihood of operand waits increases, an observation verified by detailed simulation. Also, in code #2 the workload is more unbalanced. (Consider, for example, the factorization of a 4×4 matrix with 2 processors: code #1, with p_i reducing rows i and $i + 2$, involves a 6:10 processor ratio of floating-point vector operations; code #2, with p_i reducing rows $2i - 1$ and $2i$, produces a 4:12 ratio).

It should be noted from Table 1 that efficiency (~ speedup) is higher for code #1 but execution rate is lower. It would seem that execution rate is the more significant measure.

3.3. Blocked solution results

3.3.1. Timing model. The three components of the blocked factorization of a large matrix on a VMP will now be assembled into a timing model in order to study tradeoffs in implementation of the hybrid model of Fig. 1. The following study is limited to matrix sizes that are multiples of 64, consistent with the microtasked solution; then Table 1 gives the time of Eq. (3) as a function of p, using code #1. Table 2 presents the simulated timings for block-level substitutions and multiplications for 64×64 blocks.

If $64/p$ is an integer, the substitutions and multiplications divide evenly among the processors. The time to reduce a 64×64 block row and column in Eq. (2) is then

$$T_r = T_F(p) + \frac{2b_r}{p} T_{BS} + \frac{b_r^2}{p} T_M \tag{14}$$

where b_r is the number of off-diagonal blocks, T_{BS} and T_M are defined in Table 2, and $T_F(p)$ is the time for microtasked factorization from Table 1. The two substitution steps are carried out concurrently, so the longer $A_{21}U_{11}^{-1}$ substitution timing T_{BS} is used in (14). Tasking between block-level operations—on order of 500–100 clocks from CAL—is ignored in this model.

Table 2
CRAY X-MP simulated timings on 64×64 blocks (conflict-free)

Operation	Clocks	MFLOPS
$A_{21} \leftarrow A_{21}U_{11}^{-1}$	$T_{BS} = 192328$	146
$A_{12} \leftarrow L_{11}^{-1}A_{12}$	$T_{FS} = 172020$	158
$A_{22} \leftarrow A_{22} - A_{21}A_{12}$	$T_M = 302037$	183

Table 3
Effects on global execution rate of microtasked versus uniprocessor factorization of diagonal blocks; timings are simulated

Matrix size	Microtasked MP		Uniprocessor	
	η	MFLOPS	η	MFLOPS
1 processor				
64	1.0	111.	1.0	111.
128	1.0	147.	1.0	147.
256	1.0	166.	1.0	166.
512	1.0	175.	1.0	175.
1024	1.0	179.	1.0	179.
2 processor				
64	0.971	218.	0.500	111.
128	0.976	287.	0.741	218.
256	0.981	326.	0.900	299.
512	0.988	346.	0.965	338.
1024	0.994	356.	0.989	354.
4 processors				
64	0.943	419.	0.250	111.
128	0.962	566.	0.498	293.
256	0.977	649.	0.771	512.
512	0.987	691.	0.921	645.
1024	0.993	711.	0.975	698.
8 processors				
64	0.855	760.	0.125	111.
128	0.930	1024.	0.300	353.
256	0.968	1286.	0.549	796.
512	0.984	1378.	0.845	1184.
1024	0.993	1422.	0.952	1364.
16 processors				
64	0.667	1186.	0.063	111.
128	0.845	1988.	0.167	394.
256	0.940	2498.	0.414	1102.
512	0.976	2735.	0.725	2032.
1024	0.990	2837.	0.909	2604.

3.3.2. Projected performance. The first two result columns of Table 3 depict the execution rates and efficiencies of factoring large matrices using Eq. (14) for every 64 rows and columns. For $n = 64$, the ratios of Table 1 apply. As n increases for a fixed p, the b_r^2 term in Eq. (14) predominates and the execution rate per processor approaches that of the blocked multiply, or 183 MFLOPS. Between these extremes, Table 3 shows a high efficiency, e.g.,

$$\eta \geqslant 0.968 \tag{15}$$

for $p = 8$, $n = 256$.

An alternative solution would be to ignore the coding complexity of the microtasked diagonal block factorization and perform instead a uniprocessor diagonal block factorization, while idling the remaining $p - 1$ processors. The substitution and multiplication would remain distributed among p processors. When $n = 256$ and $p = 8$, the last column of Table 3 shows that under these conditions the above efficiency of 0.968 decreases to 0.549; an efficiency of 0.97 now requires $n > 1024$. As $n \to \infty$, however, this solution again approaches 183 MFLOPS per processor.

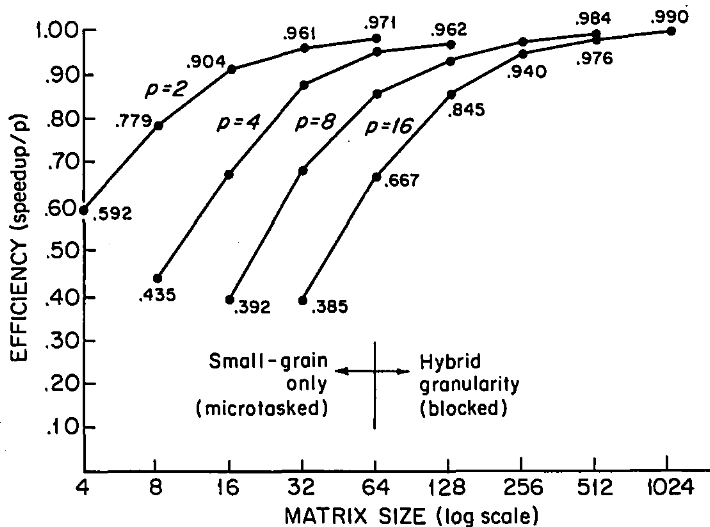

Fig. 3. Performance of hybrid code.

Figure 3 combines the performance of code #1 of Table 1 for $n \leqslant 64$ with the performance of the blocked solution of Table 3 for $n \geqslant 64$. A smooth transition is shown between a small grain solution for small problems to a hybrid solution for $n \geqslant 64$. Thus, the tasking model of Fig. 1 is dynamic as a function of n, with the large grain tasks $T_1^{(k)} \dots T_p^{(k)}$ missing for $n \leqslant 64$. This adaptibility allows exploitation of the best features of small-grain and large-grain models.

3.4. Comparisons with other parallel factorization algorithms

In [12] a factorization algorithm based on matrix-vector multiplication is given. This has the appeal of being highly modular in the Fortran level, calling on general tasking routines and efficiently-coded matrix and pivoting subroutines. A price is extracted for small problems, however. For example, in [12] a speedup of approximately 1.5 over a uniprocessor Fortran code is achieved when $n = 64$; Figure 3 indicates a speedup of 1.94 over a uniprocessor CAL code for $n = 64$ and $p = 2$. Although experimental comparative timing does not exist at this writing, an overall speedup of between 2.5 : 1 and 3 : 1 can be estimated for matrices of this size without pivoting. Asymptotically in n, overhead vanishes and both implementations would approach full machine performance. The Fortran-based code would then be desirable.

4. Conclusions

From a general algorithmic viewpoint, blocked elimination is representative of a class of MP algorithms which seek to solve tightly-coupled problems by first performing a decoupling step (diagonal block factorization) which then permits concurrent independent solution. In this case, the concurrent tasks are identical, large, and highly vectorizable—ideal for VMP architectures. If the decoupling step is a small fraction of overall computation, it is worth considering, as depicted in Table 3 for large n, assigning it to a uniprocessor, idling the other processors, and accepting a small loss in overall efficiency.

Among other conclusions specific to the CRAY are the following.

1) A central main memory, together with rotating shared registers to pass addresses and counters, will support 8–16 tightly-coupled processors in small linear algebra applications.

2) Vector instruction execution can mask interprocessor task control communication, removing signalling as a source of overhead. This probably requires use of a low-level (assembly) language to achieve concurrency of the control and numerical functions.

3) Performance of tightly-coded uniprocessor codes may suffer dramatically from operand waits not anticipated in the uniprocessor version. Codes involving simpler task phasing may be better (such as Code #1 of Table 1).

4) As Table 1 shows, speedups can be misleading representation of performance. Absolute performance is a better measure.

Acknowledgement

The author is indebted to Paul Summers and Ken Elliott for development of the CRAY X-MP simulator. The algorithm research was supported by the Air Force Office of Scientific Research under Grants 80--158 and 84-0096, the National Aeronautics Space Administration (Ames Research Center) under Grant NCC2-201, and Los Alamos National Laboratory.

References

[1] D.A. Calahan, Influence of task granularity on vector multiprocessor performance, 1984 Intl. Conf. on Par. Proc., Bellaire, MI, August, 1984, pp. 278–284.

[2] T. Jordan, and K. Fong, Some linear algebraic algorithms and their performance on the CRAY-1, Report CA-6774, Los Alamos National Laboratory, June, 1977.

[3] D.A. Calahan, High performance banded and profile equation solvers for the CRAY-1; I. The unsymmetric case, report #160, Systems Engineering Laboratory, University of Michigan, February, 1982.

[4] I.Y. Bucher, B.L. Buzbee, and P.O. Frederickson, Experiments in parallel processing a large scientific code, Proc. 1981 Intl. Conf. on Parallel Processing, pp. 166–167, August, 1981.

[5] Alan George, *Computer Solution of Large Sparse Positive Definite Systems*, (Prentice Hall, 1981).

[6] R.E. Lord, Solving linear algebraic equations on a MIMD computer, International Conf. on Parallel Processing, Bellaire, MI, 1980.

[7] D.A. Calahan, Tasking studies solving a linear algebra problem on a CRAY-class multiprocessor, Report SARL #2, Department of Electrical and Computer Engineering, University of Michigan, December, 1983.

[8] Cray XMP Series Mainframe Reference Manual, Cray Research, Inc., November 1982.

[9] Final Report, NASF Feasibility Study, NASA Report NAS2-9897, Ames Research Center, Moffett Field, CA.

[10] D. Gajski, D. Kuck, D. Laurie, and A. Sameh, Construction of a large scale multiprocessor, Report IUCDCS-R-83-1123, Computer Science Dept., University of Illinois, February, 1983.

[11] D.A. Calahan and K.E. Elliott, Memory conflict simulation of a many-processor CRAY architecture, Part I: A CRAY X-MP study, Report SARL #6, Department of Electrical Engineering and Computer Science, University of Michigan, March, 1985.

[12] S.C. Chen, J.J. Dongarra, and C. Hsuing, Multiprocessing linear algebra algorithms on the CRAY X-MP-2; Experiences with small granularity, Report ANL/MCS-TM-24, Mathematics and Computer Science Division, Argonne National Laboratory, February, 1984.

[13] BENCHLIB (preliminary documentation), Cray Research, Inc., 1984.