Multitasking the conjugate gradient on the CRAY X-MP/48

by

Gérard Meurant
Multitasking the conjugate gradient

on the CRAY X-MP/48

Gerard MEURANT

Centre d'Etudes de Limeil-Valenton

BP 27

94190 Villeneuve St Georges, France

The reproduction of this report was supported by the United States Army Research Office under Contract No. DDAG 2983-K-0124.
Abstract

We show how to efficiently implement the preconditioned conjugate gradient method on a four processors computer CRAY X-MP/48.

We solve block tridiagonal systems using block preconditioners well suited to parallel computation.

Numerical results are presented that exhibit nearly optimal speed up and high Mflops rates.
In this paper we show how to efficiently implement the preconditioned conjugate gradient method on the CRAY X-MP/48 using the four processors in parallel. We consider block tridiagonal linear systems that arise from the discretization of partial differential equations. Let

\[ Ax - b \]

be such a linear system, where

\[
A = \begin{bmatrix}
D_1 & A_2^T & \cdots & A_2 \\
A_2 & D_2 & \cdots & A_3 \\
\vdots & \vdots & \ddots & \vdots \\
A_{n-1} & D_{n-1} & \cdots & D_n \\
A_n & D_n & \cdots & \cdots
\end{bmatrix}
\]

A being a symmetric M-matrix.

In the prototype two dimensional model problem matrices \( D_i, i=1,\ldots,n \) are tridiagonal and matrices \( A_i, i=2,\ldots,n \) are diagonal.

We use the preconditioned conjugate gradient (PCG) method to solve this linear system. But if this method has been used successfully on vector computers, see for instance /10/, it is not well suited to parallel computation because of the data dependancies in the algorithm. We will show, in a similar way than /12/,/13/ how PCG can be modified to reduce the dependancies and how to get a more parallel algorithm. Another problem is to devise a preconditioner well adapted to parallel computation.
To solve this problem we will use modifications of the block preconditioners introduced in /4/, as vectorized in /10/.

The outline of the paper is as follows:

- In section 2 we briefly recall the main features of the CRAY X-MP/48 and show how to synchronize the processors.

- Section 3 gives a computational variant of the method introduced in /12/, /13/ to reduce data dependencies in PCG.

- Section 4 defines, firstly for two and then for four processors, parallel block preconditioners well adapted to the X-MP/48.

- In section 5 we present numerical results for a model problem. With two processors we get a nearly optimal speed up but with four processors memory conflicts reduce the potential speed up.
2. Overview of CRAY X-MP/48

CRAY X-MP/48 is a powerful computer with four tightly coupled processors sharing a large memory. Each processor is a vector machine with a CRAY 1 like architecture, but many improvements have been done that give a much more efficient computer. Each processor has four accesses to memory including two for vector loads and an Improved chaining mechanism allowing to exploit the full potential of the functional units.

On many vectorized codes one CRAY X-MP processor is twotothreetimes faster than the CRAY 1 although the clock period is only 1.33 times faster. The maximum performance of the four processors altogether is 840 Mflops.

The most novel feature of CRAY X-MP is that the four processors can be used simultaneously in a single job. Expression of parallelism is done inside the Fortran codes through calls to a multitasking library.

There are some low level primitives that allow

1) to start a task and eventually to wait for its completion (TSKSTART, TSKWAIT)

2) to define critical regions that must be executed by only one processor at a time (LOCKON, LOCKOFF)

3) to synchronize processors waiting for some events to happen (EVPOST, EVWAIT, EVCLEAR).

For a more detailed description of these routines see /6/, /2/, /11/.

Here we just want to describe how we can synchronize several processors using these primitives.

The Fortran list of subroutine SYNC is given in figure 1 below, /2/.
SUBROUTINE SYNC(ID)

DIMENSION ID(6)

CALL LOCKON(ID(3))
NEVN=ID(6)
ID(5)=ID(5)-1
IF(ID(5).EQ.0)THEN
   ID(5)=ID(4)
NEXEVN=3-NEVN
ID(6)=NEXEVN
CALL EVCLEAR(ID(NEXEVN))
CALL EVPOST(ID(NEVN))
CALL LOCKOFF(ID(3))
ELSE
CALL LOCKOFF(ID(3))
CALL EVWAIT(IDNEVN))
ENDIF

END

Figure 1
ID is a 6 words array that has been initialized before. ID(1) and ID(2) are events which are used in turn, ID(3) is the lock of the critical region, ID(4) is the number of tasks to be synchronized, ID(5) is the number of tasks not yet arrived at the synchronization point and ID(6) is the event used for synchronizing.

We used two events in turn because it is safer to have different events used in consecutive synchronization points, see /2/.

A synchronization barrier on the X-MP/48 is done by n processors, $2 \leq n \leq 4$, calling SYNC with the same array ID (which must be in common in the calling program). There is an overhead that comes with the synchronization, this simpliesthat the code must be large enough between two synchronization barriers for the computation not to be overhead dominated, see /3/. Previous attempts to use PCG on the CRAY X-MP were partly unsuccessful because of that problem /11/.
3. The conjugate gradient algorithm

The preconditioned conjugate gradient (PCG) method for solving a linear system of equations is as follows:

**Algorithm PCG**

Let \( x^0 \) be given, \( r^0 = b - A x^0 \) and \( p^{-1} \) arbitrary.

For \( k=0,1,\ldots \) until convergence, perform the steps:

\[
M z^k = r^k
\]

\[
\beta_k = (M z^k, z^k) / (M z^{k-1}, z^{k-1}), \quad \beta_0 = 0
\]

\[
p^k = z^k + \beta_k p^{k-1}
\]

\[
\alpha_k = (M z^k, z^k) / (A p^k, p^k)
\]

\[
x^{k+1} = x^k + \alpha_k p^k
\]

\[
r^{k+1} = r^k - \alpha_k A p^k
\]

\( M \) is a positive definite preconditioning matrix.

PCG is very well suited for vector computers if the preconditioner \( M \) is chosen carefully, see /10/ and the references therein (see also /1/).

Previous work on the use of PCG on parallel computers was done in /9/. Unfortunately there is a very low degree of parallelism in PCG as all the steps arise in a very sequential way, leading to many synchronization points and a large overhead. The two
Scalar products cannot be done in parallel and we need \( p^k \) to compute \( \alpha_k \). Regardless of \( M z^k = r^k \) the only things that can be done in parallel (at least with an acceptable granularity for the given computer) are the computations of \( x^{k+1} \) and \( r^{k+1} \).

Of course it can be said that usually, most of the time is spent in the solution of \( M z^k = r^k \) and that is does not really matter that the remaining parts are done in sequential mode. But remember that if only 10% of the algorithm is not parallel then the speed up cannot be greater than 1.82 with two processors and 3.1 with four processors. Hence it is interesting to try to increase the parallelism of PCG. Some proposals in this way have been made by Saad/12/ and Van Rosendale/13/ without numerical results. The idea is based on the following remark: in exact arithmetic we have, see /5/:

\[
(M z^i, z^j) = 0 \quad \forall i, j \neq j
\]

But

\[
z^{k+1} - z^k = -\alpha_k \quad M^{-1} A \quad p^k
\]

so

\[
M z^{k+1} - M z^k = -\alpha_k \quad A \quad p^k
\]

therefore using orthogonality

\[
(M z^{k+1}, z^{k+1}) = \alpha_k^2 (M^{-1} A \quad p^k, A \quad p^k) - (M z^k, z^k)
\]

This implies that (theoretically), we can compute \( (M z^{k+1}, z^{k+1}) \) before computing \( z^{k+1} \). The algorithm can then be conveniently recast in the following form:
Let \( x^0 \) be given, \( r^0 = b - A x^0, M z^0 = r^0, p^0 = z^0, s_0 = (r^0, z^0) \).

For \( k=0,1,... \) until convergence perform the steps

1) \( M v^k = A p^k \)

2) compute \( (v^k, A p^k), (A p^k, p^k) \)

3) \( \alpha = s_k / (A p^k, p^k) \)

\[ s_{k+1} = \alpha^2 (v^k, A p^k) - s_k \]

\[ \beta_{k+1} = s_{k+1} / s_k \]

4) \( x^{k+1} = x^k + \alpha_k p^k \)

\( r^{k+1} = r^k - \alpha_k A p^k \)

\( z^{k+1} = z^k - \alpha_k v^k \)

\( p^{k+1} = (z^k - \alpha_k v^k) + \beta_{k+1} p^k \)

In this algorithm there is one more vector to store but the parallelism has been greatly increased. Regardless of step 1), both scalar products can be computed in parallel and after the scalar step 3) the four vectors \( x, r, z, p \) can be split into as many subvectors as the number of available processors, so step 4) is fully parallel.

One may ask why we do not make the change of variable \( A' = M^{-1}A \) and \( b' = M^{-1}b \). This can
savetheneedtocomputevectorz. We are keeping z because the convergence criteria we used in PCG is $(Mz^k, z^k)/(Mz^0, z^0) \leq \varepsilon$ and we want to keep exactly the same one in the modified algorithm.

The main drawback of the last algorithm is its numerical instability. $S_k$ is supposed to tend to zero as does $\alpha_k$ and rounding errors frequently give a negative value of $s_{k+1}$ for some $k$. Then the algorithm breaks down.

However there is a way to fix this problem, the trick being to use $s_{k+1}$ as a predictor for the true value of the scalar product and to correct it, recomputing $(r^{k+}, z^{k+1})$, at the end of the iteration.
The modified algorithm \textbf{MPCG} becomes:

Let $x^0$ be given, $r^0 = b - A \cdot x^0, z^0 = r^0, p^0 = z^0$.

For $k=0,1,\ldots$ until convergence perform the steps

1) \[ M v^k = A p^k \]

2) \[ \text{compute } (v^k, A p^k), (A p^k, p^k), (r^k, z^k) \]

3) \[ \alpha_k = (r^k, z^k) / (A p^k, p^k) \]

\[ s_{k+1} = \alpha_k^2 (v^k, A p^k) - (r^k, z^k) \]

\[ \beta_{k+1} = s_{k+1} / (r^k, z^k) \]

4) \[ x^{k+1} = x^k + \alpha_k p^k \]

\[ r^{k+1} = r^k - \alpha_k A p^k \]

\[ z^{k+1} = z^k - \alpha_k v^k \]

\[ p^{k+1} = (z^k - \alpha_k v^k) + \beta_{k+1} p^k \]

There is only one more scalar product in this algorithm.

Although we are not able to theoretically prove its stability, it work quite well for
all the examples we tried.

We usually get the same number of iterations as for the standard algorithm.

We will see later on how the computations are organized.
4. Parallel preconditioning

In /4/ block preconditioning was introduced (see also /1/) in which $M$ was chosen as follows:

Let $A$ be a block diagonal matrix with diagonal blocks $\Delta_i$ $i=1,...,n$ then

$$M = (A + L) \Delta^{-1} (A + L^T)$$

where $L$ is the block lower triangular part of $A$.

Tridiagonal matrices $\Delta_i$ are computed recursively by

$$A_1 = D_1$$

$$\Delta_i = D_i - A_i \Omega_{i-1}(1) A_i^T \quad 2 \leq i \leq n$$

$\Omega_i(j)$ is a symmetric banded matrix with $2j+1$ diagonals whose elements within the band are the same as those of $\Delta_i^{-1}$.

This preconditioner is not vectorizable, therefore in /10/ when solving $M y = c$, $\Delta_i^{-1}$ was replaced by $\Omega_i(3)$. But even if now completely in vector mode, this preconditioner is not parallel as there remains two block recursions when solving $M z = r$. We now show how to modify this preconditioner for use on a parallel computer with a few vector processors. For the sake of simplicity let us begin with two processors and assume $n$ is even (but there is no loss of generality).

We use the so called "twisted factorization", see /7/, /8/.

Mistaken in the form

$$M = \Pi \Delta^{-1} \Pi^T$$
where

\[
\Pi = \begin{bmatrix}
\Delta_1 & 0 & & & \\
A_2 & \Delta_2 & 0 & & \\
& \ddots & \ddots & \ddots & \\
& & A_n & \Delta_n & A^T_{n+1} \\
& & & A_n^{n+2} & \ddots \\
& & & & \ddots & \ddots \\
& & & & & A_{n-1} \quad A_n \\
& & & & & 0 \quad \Delta_n
\end{bmatrix}
\]

When doing the product one can easily see that the resulting matrix is block tridiagonal, that is to say there is no block fill in.

Matrices \( \Delta_1 \) are computed by

\[
\Delta_1 = D_1
\]

\[
\Delta_i = D_i - A_1 \Omega_{i-1}(1) A^T_i, \quad 2 \leq i \leq n/2
\]

\[
\Delta_n = D_n
\]
\[ \Delta_i = D_i - A_{i+1}^T \Omega_{i+1}(1) A_{i+1}, \ i=n, \ldots, \ n/2-1 \]

One can remark that we do not get these formulas by direct identification, in fact we neglect a coupling term in the computation of \( \Delta_{n/2} \), but as we only need an approximation and as the numerical examples will show, this has no influence on the results. Hence the computation of \( \Delta_i \) can be done in two completely independent pieces.

The existence of the decomposition may be proven using the same techniques as in /4/ and /10/.

The solve of \( My = c \) is parallel as well. In order to vectorize, each time we have to solve a tridiagonal system, we replace \( \Delta_{i-1} \) by \( \Omega_i(3) \) as in /10/. Hence with obvious block notations, the solution \( y \) is given by

1) \[ w_1 = \Omega_1(3) c_1 \]

\[ w_i = \Omega_i(3) (c_i - A_i w_{i-1}), \ i=2, \ldots, n/2-1 \]

2) \[ w_n = \Omega_n(3) c_n \]

\[ w_i = \Omega_i(3) (c_i - A_{i+1}^T w_{i+1}), \ i=n-1, \ldots, n/2+1 \]

3) \[ w_{n/2} = \Omega_{n/2}(3) (c_n - A_{n/2} w_{n/2} - A_{n/2+1}^T w_{n/2+1}) \]

\[ y_{n/2} = w_{n/2} \]
4) \[ y_i = w_i = \Omega_i(3) A_{i+1}^T y_{i+1}, \quad i = n/2 - 1, \ldots, 1 \]

\[ y_i = w_i - \Delta_i(3) A_i y_{i-1}, \quad i = n/2 + 1, \ldots, n \]

It is obvious that steps 1) and 2) can be executed in parallel as well as 4) and 5).

Only step 3) requires the synchronization of both processors.

This method is similar to the dissection techniques used in /9/, except we are considering block methods instead of point ones.

Let us now look at how we can organize the computation in MPCG. Suppose that both processors are synchronized at the beginning of 1) (although this requirement can be weakened).

Processor 1 computes:

\[ c_i = CA \quad p^k_i, \quad i = 1, \ldots, n/2 \]

\[ w_i, \quad i = 1, \ldots, n/2 - 1 \]

\[ s_1^1 = \sum_{i=1}^{n/2} (A \quad p^k_i) \]

\[ s_1^2 = \sum_{i=1}^{n/2} (r^k_i) \]

Processor 2 computes in parallel:

\[ 15 \]
\[ c_i = [A \ p^k]_i, \quad i=n/2+1,\ldots,n \]

\[ w_i, \quad i=n,\ldots,n/2+1 \]

\[ \begin{align*}
1 & = \sum_{i=n/2+1}^{n} ([A \ p^k]_i, [p^k]_i) \\
2 & = \sum_{i=n/2+1}^{n} ([r^k]_i, [z^k]_i)
\end{align*} \]

Then both processors are synchronized, each one calling routine \textit{SYNC} of section 2 and processor 1 computes \([v^k]_{n/2}\) (note that in \textbf{a more} refined organization, after synchronization, each processor \textbf{may} compute one \textbf{half} of \([v^k]_{n/2}\)).

After this sequential step, \textbf{processor 1} computes:

\[ [v^k]_i, \quad i=n/2-1,\ldots,1 \]

\[ s_1 = \sum_{i=1}^{n/2} ([v^k]_i, [p^k]_i) \]
Processor 2 computes in parallel:

\[
[v^k]^i, \quad i = n/2+1, \ldots, n
\]

\[
S_2^3 = \sum_{i=n/2+1}^{n} ([v^k]^i, [p^k]^i)
\]

Then both processors are synchronized and after the barrier each processor is able to compute its own copy of:

\[
s_1^1 = s_1^1 + s_2^1
\]

\[
s_2^2 = s_1^2 + s_2^2
\]

\[
s_3^3 = s_1^3 + s_2^3
\]

\[
\alpha_k, \beta_k, s_{k+1}, \beta_{k+1}
\]

This is done to avoid one useless barrier and more overhead.

Having computed the scalars, Processor 1 computes

\[
[z^{k+1}]_i, [z^{k+1}]_i, [z^{k+1}]_i, [p^{k+1}]_i, i=1, \ldots, n/2
\]

and Processor 2 computes

\[
[x^{k+1}]_i, [z^{k+1}]_i, [z^{k+1}]_i, [p^{k+1}]_i, \quad i = n/2+1, \ldots, n
\]

This ends one iteration.
As the stopping criteria is $(r^k, z^k) / (r^0, z^0) < \varepsilon$, the test can be done by both processors, if the test is satisfied Processor 2 executes a return, Processor 1 waits for Processor 2 doing a TSKWAIT.

One can see that except for solving for $[v^k]_{n/2}$ everything is done in parallel and there is only 3 synchronization barriers in one 'iteration.

We will denote this method using two processors by INV2P.

Let us now look at what we can achieve with four processors.

The preconditioner is taken as

$$M = \theta A^{-1} \theta^T$$

where $\theta$ has the following structure.
\[
\Theta = \begin{bmatrix}
L_1 & N_T^T & 0 & 0 \\
0 & L_2 & 0 & 0 \\
0 & N_3 & L_3 & N_4^T \\
0 & 0 & 0 & L_4^T \\
\end{bmatrix}
\]

\[
L_1 = \begin{bmatrix}
\Delta_1 & A_2 & \Delta_2 \\
& \ddots & \ddots & \ddots \\
& & \Delta_n & A_n \\
& & & \Delta_n \\
\end{bmatrix}
\]

\[
N_2^T = \begin{bmatrix}
0 & \cdots & 0 \\
0 & \cdots & 0 \\
\cdots & \cdots & \cdots \\
0 & \cdots & 0 \\
\end{bmatrix}
\]

\[
L_2^T = \begin{bmatrix}
\frac{A_{n+1}}{4} & A_{n+2}^T \\
& \frac{A_{n+2}}{4} \\
& & \frac{A_{n+1}}{2} \\
& & & \frac{A_n}{2} \\
\end{bmatrix}
\]

\[
N_3 = \begin{bmatrix}
0 & \cdots & A_{n+1} \\
0 & \cdots & 0 \\
\cdots & \cdots & \cdots \\
0 & \cdots & 0 \\
\end{bmatrix}
\]

\[
L_3 = \begin{bmatrix}
\frac{A_{n+1}}{2} & A_{n+2} \\
& \frac{A_{n+2}}{2} \\
& & \frac{A_{3n}}{4} \\
& & & \frac{A_{3n}}{4} \\
\end{bmatrix}
\]

\[
N_4^T = \begin{bmatrix}
\frac{A_{3n+1}}{4} & \cdots & 0 \\
0 & \cdots & \cdots \\
\cdots & \cdots & \cdots \\
0 & \cdots & 0 \\
\end{bmatrix}
\]
Here $\Theta$ has the following block pattern, points showing blocks where there are non-zero elements.
Matrices $\Delta_i$ are computed by:

$$A_1 = D_1$$

$$\Delta_i = D_i = A_i \Omega_{i-1}(1) A_i^T, \quad 2 \leq i \leq n/4$$

$$\Delta_{n/2} = D_{n/2}$$

$$\Delta_i = D_i = A_{i+1}^T \Omega_{i+1}(1) A_{i+1}, \quad i = n/2 - 1, \ldots, n/4 + 1$$
\[ \Delta_i = D_i - A_i \Omega_{i-1}(1) A_i^T, \quad n/2 + 1 < i \leq 3n/4 \]

\[ \Delta_n = D_n \]

\[ \Delta_i = D_i - A_{i+1}^T \Omega_{i+1}(1) A_{i+1}, \quad i = n-1, \ldots, 3n/4 + 1 \]

A system \( My = c \) is solved in the following obvious way

1) \( w_{n/2} = \Omega_{n/2}(3) c_{n/2} \)

2) \( w_1 = \Omega_1(3) c_1 \)

\[ w_i = \Omega_i(3) (c_i - A_i w_{i-1}), \quad i = 2, \ldots, n/4 - 1 \]

3) \( w_i = \Omega_i(3) (c_i - A_{i+1}^T w_{i+1}), \quad i = n/2 - 1, \ldots, n/4 + 1 \)

4) \( w_i = \Omega_i(3) (c_i - A_i w_{i-1}), \quad i = n/2 + 1, \ldots, 3n/4 - 1 \)

5) \( w_n = \Omega_n(3) c_n \)

\[ w_i = \Omega_i(3) (c_i - A_{i+1}^T w_{i+1}), \quad i = n-1, \ldots, 3n/4 + 1 \]

6) \( w_{n/4} = \Omega_{n/4}(3) (c_{n/4} - A_{n/4} w_{n/4-1} - A_{n/4+1}^T w_{n/4+1}) \)

7) \( w_{3n/4} = \Omega_{3n/4}(3) (c_{3n/4} - A_{3n/4} w_{3n/4-1} - A_{3n/4+1}^T w_{3n/4+1}) \)
8) \( y_{n/4} = w_{n/4}, y_{3n/4} = w_{3n/4} \)

9) \( y_i = w_i - \Omega_i(3) A_{i+1}^T y_{i+1}, \quad i=n/4-1, \ldots, 1 \)

10) \( y_i = w_i - \Omega_i(3) A_i y_{i-1}, \quad i=n/4+1, \ldots, n/2-1 \)

11) \( y_i = w_i - \Omega_i(3) A_{i+1}^T y_{i+1}, \quad i=3n/4-1, \ldots, n/2+1 \)

12) \( y_i = w_i - \Omega_i(3) A_i y_{i-1}, \quad i=3n/4+1, \ldots, n \)

13) \( y_{n/2} = w_{n/2} - \Omega_{n/2}(3) (A_{n/2} y_{n/2-1} + A_{n/2+1}^T y_{n/2+1}) \)

Evidently when step 1) is completed, steps 2), 3), 4) and 5) can proceed in parallel, then steps 6) and 7) and again steps 9), 10), 11) and 12). Solving is completed by sequential step 13).

As with two processors, parts of the scalar products can be computed during steps 9), 10), 11) and 12).

There are two synchronizations of two processors and one global synchronization of the four processors at the end.

Note that when doing the product \( \Theta \Delta^{-1} e^T \), the resulting matrix is block tridiagonal except for two block fills in that we have neglected in the computation, therefore we must be prepared to have a larger number of iterations than in the standard algorithm. We denote this method with four processors by INV4P.

Note that here the method is different of the one in /9/, apart from the fact that we use block algorithms, because the blocks are not numbered in the same way.
5. Numerical results

In this section we give numerical results for the model problem viz. the discretization by a five point finite difference scheme of the Poisson equation

\[ -\Delta u = f \quad \text{in} \quad n\times n = 10 \times 10, \quad u = 0 \quad \text{on} \quad \Omega \]

Let \( h = 1/(n+1) \) be the stepsize. The matrices \( D_i \) of order \( n \) are

\[
D_i = \begin{bmatrix}
4 & -1 \\
-1 & 4 & -1 \\
& -1 & 4 & -1 \\
& & -1 & 4 \\
& & & -1 & 4
\end{bmatrix}
\]

and \( A_i = -I \), \( I \) being the identity matrix.

The chosen stopping criteria was \( 10^{-6} \).

We give detailed results for two cases, a small one with \( n=50 \) and a large one with \( n=150 \). Further results can be seen on Figures 2, 3, 4 and 5.

For each case we give results for the standard algorithm INV and the two parallel ones INV2P and INV4P. Results include the computer time (CRAY X-MP) in seconds, the number of iterations to reach the given criteria, the Megaflops rates and for parallel methods two different values of the speed up. The first one \( S_1 \) is the ratio of times for the execution of the parallel code on one processor and on several
processors, the second one $S_2$ is the ratio of the time for the standard algorithm INV to the time of the parallel algorithm. It is clear that $S_1$ measures the ability of the computer (both hardware and software) to execute a code on several processors. $S_2$ is more algorithm oriented and measures what we can gain over what was thought to be the best sequential algorithm.

<table>
<thead>
<tr>
<th>method</th>
<th>time</th>
<th>Mflops</th>
<th>iter</th>
<th>$S_1$</th>
<th>$S_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>INV</td>
<td>0.0199</td>
<td>108.86</td>
<td>17</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>INV2P</td>
<td>0.0115</td>
<td>187.55</td>
<td>19</td>
<td>1.86</td>
<td>1.73</td>
</tr>
<tr>
<td>INV4P</td>
<td>0.0127</td>
<td>220.7</td>
<td>22</td>
<td>2.75</td>
<td>1.56</td>
</tr>
</tbody>
</table>

$n=50$

**Table 1**

All time measurements have been done in dedicated mode using the real clock time. They do not include the computation of the preconditioner.

The first thing to note in this small problem is that it pays more to use two processors. The loss due to the increasing number of iterations and the larger overhead (as the vector lengths are two times shorter and there are four processors to synchronize) give a poor result with four processors, but note that the situation changes rapidly as $n$ grows.
Table 2 shows that for large problems it does pay to use four processors although the speed up is not as great as we have expected. From different experiments see /2/, we think that this problem comes from memory contention during vector operations. When we run INV4P on two processors we get a speed up larger than 1.9 indicating that the algorithm is well parallelized, but the contention between processors is larger with four processors than with two.

Figures 2, 3 4 and 5 give the Mflops rate, the speed up S1, the computing time and the number of iterations versus n (the order of A being n²).

It is clear that with four processors we have not reached the full potential of the machine and that with larger problems we can go beyond 420 Mflops but clearly the speed up is bounded. The situation for computing time is not as good because of the increase in the number of iterations going from two to four processors.
Figure 4

TIME (SEC.)
6. Conclusions

We have shown how to efficiently use the preconditioned conjugate gradient method on the CRAY X-MP/48.

With two processors we reach almost the maximum efficiency we can.

Using the four processors the situation is a little bit worst, firstly speed up is bounded by memory contention and secondly we need parallel preconditioners giving a better rate of convergence as it seems difficult to generalize INV4P to more than four processors.

Despite these facts, for large problems, we get interesting improvements over the sequential approach.

Acknowledgements

The author is indebted to CRAY Research France for their help during these computations, special thanks to Alex AZAR. Many thanks also to my colleague Jacques DAVID from CEA.
References

/1/ O. AXELSSON
Incomplete block matrix factorization preconditioning methods. The ultimate answer?
J. of comp. and appl. math. vol 12, 13 (1985) pp 3-18

/2/ Y. CHAUVET, J. DAVID & G. MEURANT
Experiences numeriques sur le CRAY X-MP/48 (in French).
Note CEA (1985)

/3/ S. CHEN, J.J. DONGARRA & C.C. HSIUNG
Multiprocessing linear algebra algorithms on the CRAY X-MP/2: experiences with small granularity

/4/ P. CONCUS, G. H. GOLUB & G. MEURANT
Block preconditioning for the conjugate gradient method

/5/ P. CONCUS, G.H. GOLUB & D.P. O' LEARY
A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations
in Sparse Matrix Computations, pp 309-332

/6/ CRAY Multitasking User's Guide SN-0222

/7/ D. FISCHER, G. H. GOLUB, O. HALD, C. LEIVA & O. WIDLUND
On Fourier-Toeplitz methods for separable elliptic problems

/8/ T.L. JORDAN
A guide to parallel computations and some CRAY 1 experiences
in Parallel Computations pp 1-50

/9/ A. LICHNEWSKY
Some vector and parallel implementations of preconditioned conjugate gradient algorithms
in High speed Computations pp 343-359

/10/ G. MEURANT
The block preconditioned conjugate gradient method on vector computers
BIT v24 (1984) pp 623433

/11/ G. MEURANT
Numerical experiments for the preconditioned conjugate gradient method on the CRAY X-MP/2
/12/ Y. SAAD

Practical use of polynomial preconditioning for the conjugate gradient method


/13/ J. VAN ROENDEALE

Minimizing inner product data dependencies in conjugate gradient iteration

ICASE NASA Langley Research Center 372178 (1983)