The
Connection Machine
System

# Scientific Subroutines

**Version 5.1**
**June 1989**

**Thinking Machines Corporation**
**Cambridge, Massachusetts**

First printing, June 1989



Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1214
(617) 876-1111

# Contents

# About the Connection Machine Scientific Software Library

## Introducing the CM Scientific Software Library

The CM Scientific Software Library is a collection of routines that support scientific computing, especially numerical analysis. Two routines from the planned library are included as part of Paris, version 5.1. These are Fast Fourier Transformation (FFT) for complex numbers and matrix multiplication for either real or complex numbers. This supplement describes these two routines.

As more numerical and scientific Connection Machine software routines become available, they will be released as a library. It will then be possible optionally to link this library with Paris or with any of the high-level Connection Machine languages, C*, *Lisp, and CM Fortran. At that time, these routines will no longer be incorporated into Paris proper.

## CMSSL Naming Conventions

All Paris-level operations destined to become part of the CMSSL are given the prefix **CMSSL:** in place of the standard **CM:** Paris instruction prefix.

Following the **CMSSL:** library prefix, a set of operation prefixes identifies the types of operands expected by each routine. In order to conform to standard math library conventions, these prefixes are slightly different from those used for normal Paris instructions.

Data Type Name Prefixes

| Prefix | Meaning |
|---|---|
| **s** | single-precision floating point |
| **d** | double-precision floating point |
| **c** | single complex (single-precision real and imaginary parts) |
| **z** | double complex (double-precision real and imaginary parts) |

For example, **s-matrix-multiply** takes single-precision operands while **c-matrix-multiply** takes single complex operands.

If it is possible for a particular kind of routine to yield a result different in data type from its operand(s), then an additional operation prefix (the first) identifies the result type. For example, **c-s-fft** (not yet available) yields a single real result but expects a single complex operand. In contrast, **c-c-fft** yields a single complex result from a single complex operand.

As with Paris proper, instruction names and the names of constants are documented in Lisp syntax. To convert an instruction name to C or Fortran, change any colon or hyphen to an underscore. For example, **CMSSL:fft-setup** becomes **CMSSL_fft_setup** in the C or Fortran interface. To convert a predefined constant operand name to C or Fortran, change any leading colon to **CMSSL_**. For example **:send-order** becomes **CMSSL_send_order**.

## Supplement Overview

This supplement is organized as is the Paris Dictionary. Entries describing each routine are ordered alphabetically, first by primary operation name and second by the type prefix.

**CMSSL:deallocate-fft-setup**
This frees front-end and CM memory used by an FFT setup descriptor.

**CMSSL:c-c-fft**
This calculates the discrete Fourier transform of a complex number and returns a complex result. A Fast Fourier Transform (FFT) algorithm is used.

**CMSSL:c-fft-setup**
This computes information needed to perform a complex FFT.

**CMSSL:c-matrix-multiply**
This performs matrix multiplication over three single-precision complex numbers.

**CMSSL:s-matrix-multiply**
This performs matrix multiplication over three single-precision floating-point numbers.

## Known Restrictions

The Scientific Software routines included in Paris Version 5.1 bear the following restrictions.

### No Back-Compatibility Mode

The CM Scientific Software does not support back-compatibility mode. Consequently, no CMSSL routines may be called from C*, Version 5.1 or earlier, because these C* versions must be run in back-compatibility mode.

### Floating-Point Hardware Required for FFT

The FFT routines are designed exclusively for Model CM2 Connection Machines with floating-point hardware.

### No CM Fortran Complex Matrix Multiply

In the current release, the **CMSSL:c-matrix-multiply** routine is not available when calling Paris from CM Fortran. CM Fortran does not yet support complex numbers.

## Documentation Discrepancies

One routine and a set of constants are, in Version 5.1, implemented under names different from those documented in this supplement. The implementation of Version 5.2 will match the current documentation.

### CMSSL:deallocate-fft-setup

This is currently implemented as **CMSSL:deallocate-setup**. In Version 5.2, the implemented name will match the documented name, **CMSSL:deallocate-fft-setup**.

### FFT Constant Types

The names of the C/Paris constants defined as allowable values to the *setup*, *ops*, *source-bit-order*, *dest-bit-order*, *source-cm-order*, *dest-cm-order*, and *scale* arguments to **CMSSL:c-c-fft** are all documented to begin with **CMSSL_**. In Version 5.1, they instead begin with **FFT_**. For example the *ops* vector is documented to expect elements named **CMSSL_f_xform**, **CMSSL_i_xform**, and **CMSSL_nop**. In fact, valid values are **FFT_f_xform**, **FFT_i_xform**, and **FFT_nop**. In Version 5.2 these constant names will all begin with **CMSSL_**.

The **CMSSL:c-c-fft** *dest-bit-order* and *source-bit-order* vector arguments are documented to take a value of **CMSSL_default** to specify default ordering for the corresponding axis. In fact, the implementation requires that the constant name **FFT_default_124** be used. This will not be the case in the implementation of Version 5.2.

# CMSSL Dictionary

# DEALLOCATE-FFT-SETUP

Deallocates a front-end setup descriptor that has been used to prepare information for execution of an FFT routine.

In a future release, this will be part of a Scientific Software Library. For this reason, it is given the prefix CMSSL: in place of the standard CM: Paris instruction prefix.

---

**Formats** CMSSL:deallocate-fft-setup *setup-id*

Operands *setup-id* The id of the FFT setup descriptor to be deallocated.

Context This is a front-end operation. It does not depend on the value of the *context-flag*.

---

This routine may be used to remove an FFT setup descriptor when it is no longer needed. The setup-id argument must have been obtained by a call to one of the CMSSL:fft-setup routines.

An fft setup descriptor occupies memory both on the front end and on the Connection Machine. It is therefore wise to free this space by calling CMSSL:deallocate-fft-setup after completion of all FFT routines that use the specified setup descriptor.

# C-C-FFT

The Discrete Fourier Transform of the complex source field is calculated using a Fast Fourier Transform (FFT) algorithm. The complex result is stored in the destination field.

A Fourier transform routine converts (possibly multidimensional) sequences between the time or space domain and the frequency domain. This type of transform has a variety of useful applications. For example, an FFT can be used to filter discrete signals, to smooth input data or output images, to interpolate or extrapolate from a given data set, to measure the correlation between two samples, or to multiply polynomials and extremely large integers.

The Fast Fourier Transform is called a fast transform because it exhibits $O(N \log N)$ complexity, where $O$ is the order of complexity and $N$ is the length of the input sequence. By comparison, the Discrete Fourier Transform exhibits only $O(N^2)$ complexity.

In a future release, this routine will be part of the Connection Machine Scientific Software Library. For this reason, it is given the prefix CMSSL: in place of the standard CM: Paris instruction prefix.

---

**Formats** CMSSL:c-c-fft *dest*, *source*, *setup*, *ops*, *source-bit-order*, *dest-bit-order*, *source-cm-order*, *dest-cm-order*, *scale*

**Operands**

*dest* The complex destination field.

*source* The complex source field.

*setup* The setup-id. This must be a setup-id returned by CMSSL:c-fft-setup. The geometry information of the setup must be identical to that of the source and destination fields.

*ops* A front-end vector of operation identifiers. Each element specifies whether the corresponding source axis is transformed and, if so, by what method. Valid vector element values are :f-xform (CMSSL_f_xform in C; 1 in Fortran) for a forward transform, :i-xform (CMSSL_i_xfrom in C; 2 in Fortran) for an inverse transform, and :nop (CMSSL_nop in C; 0 in Fortran) for no transform.

*source-bit-order* A front-end vector of input bit orderings. Each element identifies the bit ordering of the corresponding source axis and must be either :normal or :bit-reversed. (The corresponding values are are CMSSL_normal and CMSSL_bit_reversed in C, and 0 and 1 in Fortran, respectively.)

*dest-bit-order* A front-end vector of output bit orderings. Each element identifies the bit ordering of the corresponding destination axis and must be either :normal or :bit-reversed. (The corresponding

values are are CMSSL_normal and CMSSL_bit_reversed in C, and 0 and 1 in Fortran, respectively.)

*source-cm-order* A front-end vector of input orderings. Each element declares the addressing mode of the corresponding source axis and must be one of the following: :send-order, :news-order, or :default. (The corresponding values are CMSSL_send_order, CMSSL_news_order, and CMSSL_default in C, and 1, 2, and 0 in Fortran, respectively.)

A value of :default causes the current ordering of an axis to be used.

*dest-cm-order* A front-end vector of output orderings. Each element declares the addressing mode of the corresponding destination axis and must be one of the following: :send-order, :news-order, or :default. (The corresponding values are CMSSL_send_order, CMSSL_news_order, and CMSSL_default in C, and 1, 2, and 0 in Fortran, respectively.)

A value of :default causes the current ordering of an axis to be used.

*scale* A front-end vector of output scaling methods. Each element specifies whether the corresponding destination axis is rescaled and, if so, by what method. Valid values are :noscale for no rescaling, :scale-sqrt for scaling by the inverse square root of the FFT, and :scale-n for scaling by the inverse of the size of the FFT. (The corresponding values are CMSSL_noscale, CMSSL_scale_sqrt, and CMSSL_scale_n in C, and 0, 1, and 2 in Fortran, respectively.)

Overlap The *source* field must be either disjoint from or identical to the *dest* field. Two complex fields are identical if they have the same address and the same format. FFT performance is slightly better if the two fields are identical.

Context This operation is unconditional. It does not depend on the *context-flag*.

---

**Definition** For every virtual processor $k$ in the *current-vp-set* do
$dest[k] \leftarrow FFT(source[k])$

The Discrete Fourier Transform of the *source* field is stored in the *dest* field. A multi-dimensional transform is computed by performing the transform across each dimension in sequence.

The source and destination fields must either belong to the same VP set or to VP sets of identical shape and size.

The *ops*, *source-bit-order*, *dest-bit-order*, *source-cm-order*, *dest-cm-order*, and *scale* arguments are one-dimensional front-end arrays. The length of each is equal to the rank of the setup geometry.

By convention, a Fast Fourier Transform operation reverses the order of the data bits when storing the result in the destination. The vectors *source-bit-order* and *dest-bit-order* specify whether the source and destination data are treated as normal or as bit-reversed.

Along any given dimension of the data's geometry, the Connection Machine FFT instruction is most efficient for data arranged in send order. Many FFT applications do not depend on the order of the data elements. The *dest-cm-order* and *source-cm-order* arguments are therefore provided to permit the most efficient execution possible along each dimension.

For further details see the Thinking Machines technical report number NA87-3 entitled "Computing Fast Fourier Transforms on Boolean Cubes and Related Networks" and the upcoming Thinking Machines technical report entitled "A Cooley Tukey FFT on the Connection Machine."

The following example code demonstrates invocation of CM:c-c-fft from *Lisp and may be found on line in the directory /cm/examples/cmssl.

```
;; This example code is designed to illustrate the calling requirements
;; of the FFT initialization routines and of the FFT itself.
;;
;; Notice that the FFT routines are a part of the CMSSL package.
;;
;; The example problem is a two dimensional FFT (1024 by 1024)
;; The data is assumed to be a front-end array

(in-package '*lisp)
(defun fft_example()

;; Set up various FFT parameters from a geometry description.
;; Create the geometry. Call c-fft-setup to intialize the setup structure,
;; using fft-geometry as input.

  (let* ( (fft-geometry (cm:create-geometry '(1024 1024)))
     (fft-vpset (cm:declare-vp-set fft-geometry))
     (fft-setup (CMSSL::c-fft-setup fft-geometry)) )
   (declare (type cmssl::fft-setup fft-setup))

;; The initial pattern is specified by the data. A forward FFT transform
;; will be performed along the second dimension. Neither axis is scaled.
;; The bit-orders are :normal and the cm-orders are the default.

  (let* ((rank (cm:geometry-rank fft-geometry))
     (ops (vector :nop :f-xform))
     (source-bit-orders (vector :normal :normal))
     (dest-bit-orders (vector :normal :normal))
     (send-cm-orders (vector :default :default))
     (noscales (vector :noscale :noscale)))

;; The FFT must be computed within the fft-vpset.

   (*with-vp-set fft-vpset
    (*let ((data))
     (declare (type (pvar (complex-float)) data))
     (let ((data-loc (pvar-location data)))

;; Input the data on which the fft will be performed.
;; Exchange any initialization routine for my-initialize-data.

      (my-initialize-data data)
```

```
;;
;;Now perform FFT on data, using data as the destination too.
;;
      (CMSSL:c-c-fft data-loc data-loc fft-setup ops
      source-bit-orders dest-bit-orders
      send-cm-orders send-cm-orders noscales)
     );end let
    );end *let
   );end *with-vp-set
  );end let*
);end let*
);end defun
```

# C-FFT-SETUP

Allocates a front-end setup descriptor for use with the CMSSL:fft Fast Fourier Transform routines and returns a setup-id.

In a future release, this will be part of a Scientific Software Library. For this reason, it is given the prefix CMSSL: in place of the standard CM: Paris instruction prefix.

---

**Formats** result ← CMSSL:c-fft-setup *geometry-id*

**Operands** *geometry* A geometry-id.

**Result** The id of the newly created FFT setup descriptor.

**Context** This is a front-end operation. It does not depend on the value of the *context-flag*.

---

This routine computes information needed to perform a Fast Fourier Transform (FFT), stores it in an FFT setup descriptor, and return the setup-id.

In Lisp/Paris, a setup-id is a structure of type CMSSL:fft-setup. In C/Paris, it is a pointer to a structure of type CMSSL_fft_setup_t. In Fortran/Paris it is an integer.

The *geometry* argument must be a geometry-id returned by a call to CM:create-geometry, CM:create-detailed-geometry, intern-geometry, or intern-detailed-geometry.

The returned setup-id is a valid value for the *setup* argument to any CMSSL FFT routine if the following requirement is obeyed. The geometries of the FFT source and destination fields must be identical to that of the setup geometry.

This routine must be reinvoked whenever the geometry of an FFT source field VP set is changed. CMSSL:c-fft-setup allocates memory both on the front end and on the CM. To free this memory, use CMSSL:deallocate-fft-setup.

See the description of CM:c-c-fft for a code example demonstrating how to call CM:c-fft-setup.

# C-MATRIX-MULTIPLY

Computes matrix multiplication using three single-precision complex operands and stores the result in the last.

In a future release, this routine will be part of a Scientific Software Library. For this reason, it is given the prefix CMSSL: in place of the standard CM: Paris instruction prefix.

---

**Formats** CMSSL:c-matrix-multiply *source1*, *source2*, *dest/source3*

| | | |
|---|---|---|
| Operands | *dest* | The complex destination field. |
| | *source1* | The complex first source field. |
| | *source2* | The complex second source field. |
| | *source3* | The complex third source field. |
| Overlap | The fields *source1*, *source2*, and *dest/source3* must not overlap in any manner. | |
| Context | This operation is unconditional. It does not depend on the *context-flag*. | |

---

The calculation *dest* $\leftarrow$ *source3* + *source1* $\times$ *source2* is performed on three conforming matrices, represented as CM fields.

The operands *source1*, *source2*, and *dest/source3* must be fields of 64-bit single-precision complex values whose real and imaginary parts are 32-bit floating-point values.

All three operands may belong to separate VP sets if the geometries of those VP sets obey the following rule:

- The *source1* dimensions are $n \times m$
- The *source2* dimensions are $m \times p$
- The *dest/source3* dimensions are $n \times p$

where $n$, $m$, and $p$ are each powers of two. Otherwise, all three operands must belong to the same square VP set.

The matrix multiply is performed using Cannon's systolic algorithm, which can be summarized in three steps:

1. The *source1* and *source2* matrices are aligned so the elements in each processor have conforming indices for matrix multiplication. In terms of data motion, this implies aligning the diagonal entries of the *source1* matrix to the first column and aligning the diagonal entries of the *source2* matrix to the first row.

2. The systolic part of the algorithm involves local multiplication of *source1* and *source2* elements followed by nearest neighbor data moves that simulate the inner product.

3. The *source1* and *source2* matrices are aligned back to the original form supplied by the calling program.

In order to exploit the full potential of the floating-point hardware, a block version of the algorithm is implemented. See the Thinking Machines technical report entitled "Matrix Multiplication on the Connection Machine" for details.

The CM matrix multiplication operation performs best for square matrices and at high VP ratios.

The following code examples demonstrate invocation of CM:c-matrix-multiply from C/Paris, from Fortran/Paris, and from *Lisp. These may be found on line in the directory /cm/examples/cmssl.

```
/* Example C/Paris matrix multiply code */
#include <stdio.h>
#include <math.h>
#include <cm/paris.h>

CM_geometry_id_t  A_geom;
CM_geometry_id_t  B_geom;
CM_geometry_id_t  C_geom;

CM_vp_set_id_t    A_vpset;
CM_vp_set_id_t    B_vpset;
CM_vp_set_id_t    C_vpset;

CM_field_id_t     a;
CM_field_id_t     b;
CM_field_id_t     c;

CM_field_id_t     a2;
CM_field_id_t     b2;
CM_field_id_t     c2;

int idim[2];
int length=64;
int s=23;
int e=8;
int n;
int m;
int p;
```

```
main(ac, av)
int ac;
char **av;

{
double cmin;
double cmax;

    CM_init();

/* Dimensions of matrices are user input. */
/* source1 (A) is n times m, source2 (B) is m times r */
/* and dest/source3 (C) is n times p */

    printf("\\n Enter n, m, p\\n");
    scanf("%d %d %d", &n, &m, &p);

    idim[0] = n;
    idim[1] = m;

    printf("\\n Creating vp set for A of size %d %d", n, m);

    A_geom = CM_create_geometry(idim, 2);
    A_vpset = CM_allocate_vp_set(A_geom);
    CM_set_vp_set(A_vpset);
    a = CM_allocate_stack_field(length);
    a2 = CM_add_offset_to_field_id(a, 32);

/* Initialize A */

    CM_f_move_const_always_1L(a, 2.0, s, e);
    CM_f_move_const_always_1L(a2, 2.0, s, e);

    printf("\\n Creating vp set for B of size %d %d",m, p);

    idim[0] = m;
    idim[1] = p;

    B_geom = CM_create_geometry(idim, 2);
    B_vpset = CM_allocate_vp_set(B_geom);
    CM_set_vp_set(B_vpset);
    b = CM_allocate_stack_field(length);
    b2 = CM_add_offset_to_field_id(b, 32);
```

```
/* Initialize B */

    CM_f_move_const_always_1L(b, 1.0, s, e);
    CM_f_move_const_always_1L(b2, 1.0, s, e);

    printf("\\n Creating vp set for C of size %d %d", n, p);

    idim[0] = n;
    idim[1] = p;

    C_geom = CM_create_geometry(idim, 2);
    C_vpset = CM_allocate_vp_set(C_geom);
    CM_set_vp_set(C_vpset);
    c = CM_allocate_stack_field(length);
    c2 = CM_add_offset_to_field_id(c, 32);

/* Initilize C */

    CM_f_move_const_always_1L(c, 0.0, s, e);
    CM_f_move_const_always_1L(c2, 0.0, s, e);

    CMSSL_c_matrix_multiply(a, b, c);

    cmin = CM_global_f_min_1L(c, s, e);
    cmax = CM_global_f_max_1L(c, s, e);

    printf("\\n minimum value =%g maximum value=%g \\n", cmin, cmax);

    cmin = CM_global_f_min_1L(c2, s, e);
    cmax = CM_global_f_max_1L(c2, s, e);

    printf("\\n minimum value =%g maximum value=%g \\n", cmin, cmax);

    CM_deallocate_stack_through(c);
    CM_deallocate_geometry(A_geom);
    CM_deallocate_geometry(B_geom);
    CM_deallocate_geometry(C_geom);
}
```

```
c ** Example Fortran/Paris matrix multiply code
  program matrix

  integer a_loc, b_loc, c_loc, n, m, p
  integer a_loc2, b_loc2, c_loc2
  integer A_geom, B_geom, C_geom, idim(2)
  integer A_vpset, B_vpset, C_vpset, send

  include '/usr/include/cm/paris-configuration-fort.h'

  data length /64/
  data isig /23/
  data iexp /8/

c ** Dimensions of the matrices are user input.
c ** Source1 (A) is n times m, source2 (B) is m times r,
c ** and dest/source3 (C) is n times p.

  write(*,*) ' Please enter n, m, p'
  read(*,*) n, m, p

c ** initialize the CM.
  call CM_init()

c ** A's vp set.
  print *, ' creating vp set for A of size',n,m

  idim(1) = n
  idim(2) = m
  A_geom = CM_create_geometry(idim, 2)
  A_vpset = CM_allocate_vp_set(A_geom)
  call CM_set_vp_set(A_vpset)
  a_loc = CM_allocate_stack_field(length)
  a_loc2 = CM_add_offset_to_field_id(a_loc,32)

c ** Initialize A
  call CM_f_move_const_always_1l(a_loc, 2.0, isig, iexp)
  call CM_f_move_const_always_1l(a_loc2, 2.0, isig, iexp)

c ** B's vp set.
  print *, ' creating vp set for B of size',m,p

  idim(1) = m
  idim(2) = p
  B_geom = CM_create_geometry(idim, 2)
```

```
      B_vpset = CM_allocate_vp_set(B_geom)
      call CM_set_vp_set(B_vpset)
      b_loc = CM_allocate_stack_field(length)
      b_loc2 = CM_add_offset_to_field_id(b_loc,32)

c ** Initialize B
      call CM_f_move_const_always_1l(b_loc, 1.0, isig, iexp)
      call CM_f_move_const_always_1l(b_loc2, 1.0, isig, iexp)

c ** C's vp set.
      print *, ' creating vp set for C of size',n,p

      idim(1) = n
      idim(2) = p
      C_geom = CM_create_geometry(idim, 2)
      C_vpset = CM_allocate_vp_set(C_geom)
      call CM_set_vp_set(C_vpset)
      c_loc = CM_allocate_stack_field(length)
      c_loc2 = CM_add_offset_to_field_id(c_loc,32)

c ** Initialize C
      call CM_f_move_const_always_1l(c_loc, 0.0, isig, iexp)
      call CM_f_move_const_always_1l(c_loc2, 0.0, isig, iexp)

      call CMSSL_C_MATRIX_MULTIPLY(a_loc, b_loc, c_loc)

      print *, ' finally the result.....'

      cmin = CM_global_f_min_1l(c_loc, isig, iexp)
      cmax = CM_global_f_max_1l(c_loc, isig, iexp)

      print *, ' cmin =',cmin, ' cmax =',cmax

      cmin = CM_global_f_min_1l(c_loc2, isig, iexp)
      cmax = CM_global_f_max_1l(c_loc2, isig, iexp)

      print *, ' cmin =',cmin, ' cmax =',cmax

c ** the cleanup phase.
      call CM_deallocate_stack_through(C_loc)
      call CM_deallocate_geometry(A_geom)
      call CM_deallocate_geometry(B_geom)
      call CM_deallocate_geometry(C_geom)
      stop
      end
```

```
;;; *Lisp/Paris matrix multiply code fragment

(defun test-complex-from-paris (ax ay by)
  ;
  ; Multiply => A(ax, ay) X B(ay, by) + C(ax, by) = C
  ;
  (format t "   (format t "A is  s by  s, B is  s by  s and C is  s by  s"
   ax ay ay by ax by)

  (let* ( (A-geom (cm:create-geometry (vector ax ay) 2))
    (B-geom (cm:create-geometry (vector ay by) 2))
    (C-geom (cm:create-geometry (vector ax by) 2))
    (A-vp-set (cm:allocate-vp-set A-geom))
    (B-vp-set (cm:allocate-vp-set B-geom))
    (C-vp-set (cm:allocate-vp-set C-geom))
    (old-vp-set cm:*current-vp-set*)
    (length 64) (s 23) (e 8) a b c )

   (cm:set-vp-set A-vp-set)
   (setq a (cm:allocate-stack-field length))
   (cm:c-move-const-1l a (complex 2.0 1.0) s e)

   (cm:set-vp-set B-vp-set)
   (setq b (cm:allocate-stack-field length))
   (cm:c-move-const-1l b (complex 1.0 5.0) s e)

   (cm:set-vp-set C-vp-set)
   (setq c (cm:allocate-stack-field length))
   (cm:c-move-const-1l c (complex 0.0 0.0) s e)

   (cmssl::c-matrix-multiply A B C)

  ; clean up phase.

   (cm:deallocate-stack-through c)
   (cm:deallocate-geometry a)
   (cm:deallocate-geometry b)
   (cm:deallocate-geometry c)
  ) ; end let*.
) ; end defun.
```

# S-MATRIX-MULTIPLY

Computes matrix multiplication using three single-precision floating-point operands and stores the result in the last.

In a future release, this instruction will be part of a Scientific Software Library. For this reason, it is given the prefix CMSSL: in place of the standard CM: Paris instruction prefix.

---

**Formats** CMSSL:s-matrix-multiply *source1*, *source2*, *dest/source3*

| | | |
|---|---|---|
| **Operands** | *dest* | The floating-point destination field. |
| | *source1* | The floating-point first source field. |
| | *source2* | The floating-point second source field. |
| | *source3* | The floating-point third source field. |
| **Overlap** | The fields *source1*, *source2*, and *dest/source3* must not overlap in any manner. | |
| **Context** | This operation is unconditional. It does not depend on the *context-flag*. | |

---

The calculation *dest* $\leftarrow$ *source3* + *source1* $\times$ *source2* is performed on three conforming matrices, represented as CM fields.

The operands *source1*, *source2*, and *dest/source3* must be fields of 32-bit single-precision floating-point values.

All three operands may belong to separate VP sets if the geometries of those VP sets obey the following rule:

- The *source1* dimensions are $n \times m$
- The *source2* dimensions are $m \times p$
- The *dest/source3* dimensions are $n \times p$

where $n$, $m$, and $p$ are each powers of two. Otherwise, all three operands must belong to the same square VP set.

The matrix multiply is performed using Cannon's systolic algorithm, which can be summarized in three steps:

1. The *source1* and *source2* matrices are aligned so the elements in each processor have conforming indices for matrix multiplication. In terms of data motion, this implies aligning the diagonal entries of the *source1* matrix to the first column and aligning the diagonal entries of the *source2* matrix to the first row.

2. The systolic part of the algorithm involves local multiplication of *source1* and *source2* elements, followed by nearest neighbor data moves that simulate the inner product.

3. The *source1* and *source2* matrices are aligned back to the original form supplied by the calling program.

In order to exploit the full potential of the floating-point hardware, a block version of the algorithm is implemented. See the Thinking Machines technical report entitled "Matrix Multiplication on the Connection Machine" for details.

The CM matrix multiplication routine performs best for square matrices and at high VP ratios.

The following code examples demonstrate invocation of CM:s-matrix-multiply from C/Paris, from Fortran/Paris, and from *Lisp. These examples may be found on line in the directory /cm/examples/cmssl.

```
/* Example C/Paris matrix multiply code */
#include <stdio.h>
#include <math.h>
#include <cm/paris.h>

CM_geometry_id_t  A_geom;
CM_geometry_id_t  B_geom;
CM_geometry_id_t  C_geom;

CM_vp_set_id_t    A_vpset;
CM_vp_set_id_t    B_vpset;
CM_vp_set_id_t    C_vpset;

CM_field_id_t     a;
CM_field_id_t     b;
CM_field_id_t     c;

int idim[2];
int length=32;
int s=23;
int e=8;
int n;
int m;
int p;

main(ac, av)
int ac;
char **av;
```

```
{
double cmin;
double cmax;

    CM_init();

/* Dimensions of matrices are user input. */
/* Source1 (A) is n times m, source2 (B) is m times p, */
/* and dest/source3 (C) is n times p */

    printf("\\n Enter n, m, p\\n");
    scanf("%d %d %d", &n, &m, &p);

    idim[0] = n;
    idim[1] = m;

    printf("\\n Creating vp set for A of size %d %d", n, m);

    A_geom = CM_create_geometry(idim, 2);
    A_vpset = CM_allocate_vp_set(A_geom);
    CM_set_vp_set(A_vpset);
    a = CM_allocate_stack_field(length);
    CM_f_move_const_always_1L(a, 2.0, s, e);

    printf("\\n Creating vp set for B of size %d %d",m, p);

    idim[0] = m;
    idim[1] = p;

    B_geom = CM_create_geometry(idim, 2);
    B_vpset = CM_allocate_vp_set(B_geom);
    CM_set_vp_set(B_vpset);
    b = CM_allocate_stack_field(length);
    CM_f_move_const_always_1L(b, 1.0, s, e);

    printf("\\n Creating vp set for C of size %d %d", n, p);

    idim[0] = n;
    idim[1] = p;

    C_geom = CM_create_geometry(idim, 2);
    C_vpset = CM_allocate_vp_set(C_geom);
    CM_set_vp_set(C_vpset);
    c = CM_allocate_stack_field(length);
```

```
    CM_f_move_const_always_1L(c, 0.0, s, e);

    CMSSL_s_matrix_multiply(a, b, c);

    cmin = CM_global_f_min_1L(c, s, e);
    cmax = CM_global_f_max_1L(c, s, e);

    printf("\\n minimum value =%g maximum value=%g \\n", cmin, cmax);

    CM_deallocate_stack_through(c);
    CM_deallocate_geometry(A_geom);
    CM_deallocate_geometry(B_geom);
    CM_deallocate_geometry(C_geom);
}
```

```
c ** Example Fortran/Paris matrix multiply code
  program matrix

  integer a_loc, b_loc, c_loc, n, m, p
  integer A_geom, B_geom, C_geom, idim(2)
  integer A_vpset, B_vpset, C_vpset, send

  include '/usr/include/cm/paris-configuration-fort.h'

  data length /32/
  data isig /23/
  data iexp /8/

c ** Dimensions of the matrices are user input.
c ** Source1 (A) is n time m, source2 (B) is m times r,
c ** and dest/source3 (C) is n times r.

  write(*,*) ' Please enter n, m, p'
  read(*,*) n, m, p

c ** initialize the CM.

  call CM_init()

c ** A's vp set.

  print *, ' creating vp set for A of size',n,m

  idim(1) = n
  idim(2) = m
  A_geom = CM_create_geometry(idim, 2)
  A_vpset = CM_allocate_vp_set(A_geom)
  call CM_set_vp_set(A_vpset)
  a_loc = CM_allocate_stack_field(length)
  call CM_f_move_const_always_1l(a_loc, 2.0, isig, iexp)

c ** B's vp set.

  print *, ' creating vp set for B of size',m,p

  idim(1) = m
  idim(2) = p
  B_geom = CM_create_geometry(idim, 2)
  B_vpset = CM_allocate_vp_set(B_geom)
  call CM_set_vp_set(B_vpset)
```

```
      b_loc = CM_allocate_stack_field(length)
      call CM_f_move_const_always_1l(b_loc, 1.0, isig, iexp)

c ** C's vp set.

      print *, ' creating vp set for C of size',n,p

      idim(1) = n
      idim(2) = p
      C_geom = CM_create_geometry(idim, 2)
      C_vpset = CM_allocate_vp_set(C_geom)
      call CM_set_vp_set(C_vpset)
      c_loc = CM_allocate_stack_field(length)
      call CM_f_move_const_always_1l(c_loc, 0.0, isig, iexp)

      call CMSSL_S_MATRIX_MULTIPLY(a_loc, b_loc, c_loc)

      print *, ' finally the result.....'

      cmin = CM_global_f_min_1l(c_loc, isig, iexp)
      cmax = CM_global_f_max_1l(c_loc, isig, iexp)

      print *, ' cmin =',cmin, ' cmax =',cmax

c ** the cleanup phase.

      call CM_deallocate_stack_through(C_loc)
      call CM_deallocate_geometry(A_geom)
      call CM_deallocate_geometry(B_geom)
      call CM_deallocate_geometry(C_geom)
      stop
      end
```

```
;;; *Lisp/Paris matrix multiply code fragment

(defun test-real-from-paris (ax ay by)
;
; Multiply => A(ax, ay) X B(ay, by) + C(ax, by) = C

(format t " (format t "A is  s by  s, B is  s by  s and C is  s by  s"
   ax ay ay by ax by)

  (let* ( (A-geom (cm:create-geometry (vector ax ay) 2))
    (B-geom (cm:create-geometry (vector ay by) 2))
    (C-geom (cm:create-geometry (vector ax by) 2))
    (A-vp-set (cm:allocate-vp-set A-geom))
    (B-vp-set (cm:allocate-vp-set B-geom))
    (C-vp-set (cm:allocate-vp-set C-geom))
    (old-vp-set cm:*current-vp-set*)
    (length 32) (s 23) (e 8) a b c )

   (cm:set-vp-set A-vp-set)
   (setq a (cm:allocate-stack-field length))
   (cm:f-move-const-always-11 a 2.0 s e)

   (cm:set-vp-set B-vp-set)
   (setq b (cm:allocate-stack-field length))
   (cm:f-move-const-always-11 b 1.0 s e)

   (cm:set-vp-set C-vp-set)
   (setq c (cm:allocate-stack-field length))
   (cm:f-move-const-always-11 c 0.0 s e)

   (cmssl::s-matrix-multiply A B C)

  ; clean up phase.

   (cm:deallocate-stack-through c)
   (cm:deallocate-geometry a)
   (cm:deallocate-geometry b)
   (cm:deallocate-geometry c)
  ) ; end let*.
) ; end defun.
```