Threaded Charm++ Manual
The best contact path for licensing issues is by e-mail to kale@illinois.edu or send correspondence to:

Prof. L. V. Kale
Dept. of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, Illinois 61801 USA
FAX: (217) 244-6500
Contents

1 Motivation 4

2 Basic TCharm Programming 4
   2.1 Global Variables .................................................. 4
   2.2 Input/Output .......................................................... 5
   2.3 Migration-Based Load Balancing .................................. 5

3 Advanced TCharm Programming 6
   3.1 Writing a Pup Routine ............................................. 6
   3.2 Readonly Global Variables ....................................... 8

4 Combining Frameworks 9

5 Command-line Options 10

6 Writing a library using TCharm 10
1 Motivation

Charm++ includes several application frameworks, such as the Finite Element Framework, the Multiblock Framework, and AMPI. These frameworks do almost all their work in load balanced, migratable threads.

The Threaded Charm++ Framework, TCharm, provides both common runtime support for these threads and facilities for combining multiple frameworks within a single program. For example, you can use TCharm to create a Finite Element Framework application that also uses AMPI to communicate between Finite Element chunks.

Specifically, TCharm provides language-neutral interfaces for:

1. Program startup, including read-only global data setup and the configuration of multiple frameworks.
2. Run-time load balancing, including migration.
3. Program shutdown.

The first portion of this manual describes the general properties of TCharm common to all the application frameworks, such as program contexts and how to write migratable code. The second portion describes in detail how to combine separate frameworks into a single application.

2 Basic TCharm Programming

Any routine in a TCharm program runs in one of two contexts:

Serial Context  Routines that run on only one processor and with only one set of data. There are absolutely no limitations on what a serial context routine can do—it is as if the code were running in an ordinary serial program. Startup and shutdown routines usually run in the serial context.

Parallel Context Routines that run on several processors, and may run with several different sets of data on a single processor. This kind of routine must obey certain restrictions. The program’s main computation routines always run in the parallel context.

Parallel context routines run in a migratable, user-level thread maintained by TCharm. Since there are normally several of these threads per processor, any code that runs in the parallel context has to be thread-safe. However, TCharm is non-preemptive, so it will only switch threads when you make a blocking call, like “MPI_RECV” or “FEM_UPDATE_FIELD”.

2.1 Global Variables

By “global variables”, we mean anything that is stored at a fixed, preallocated location in memory. In C, this means variables declared at file scope or with the static keyword. In Fortran, this is either variables that are part of a COMMON block, declared inside a MODULE or variables with the SAVE attribute.

Global variables are shared by all the threads on a processor, which makes using global variables extremely error prone. To see why this is a problem, consider a program fragment like:

```c
foo=a
call MPI_RECV(...) b=foo
```

After this code executes, we might expect b to always be equal to a. But if foo is a global variable, MPI_RECV may block and foo could be changed by another thread.

For example, if two threads execute this program, they could interleave like:
At this point, thread 1 might expect $b$ to be 1; but it will actually be 2. From the point of view of thread 1, the global variable $foo$ suddenly changed its value during the call to `MPI_Recv`.

There are several possible solutions to this problem:

- Never use global variables—only use parameters or local variables. This is the safest and most general solution. One standard practice is to collect all the globals into a C struct or Fortran type named “Globals”, and pass a pointer to this object to all your subroutines. This also combines well with the pup method for doing migration-based load balancing, as described in Section 3.1.

- Never write different values to global variables. If every thread writes the same value, global variables can be used safely. For example, you might store some parameters read from a configuration file like the simulation timestep $\Delta t$. See Section 3.2 for another, more convenient way to set such variables.

- Never issue a blocking call while your global variables are set. This will not work on a SMP version of Charm++, where several processors may share a single set of global variables. Even on a non-SMP version, this is a dangerous solution, because someday someone might add a blocking call while the variables are set. This is only a reasonable solution when calling legacy code or using old serial libraries that might use global variables.

The above only applies to routines that run in the parallel context. There are no restrictions on global variables for serial context code.

### 2.2 Input/Output

In the parallel context, there are several limitations on open files. First, several threads may run on one processor, so Fortran Logical Unit Numbers are shared by all the threads on a processor. Second, open files are left behind when a thread migrates to another processor—it is a crashing error to open a file, migrate, then try to read from the file.

Because of these restrictions, it is best to open files only when needed, and close them as soon as possible. In particular, it is best if there are no open files whenever you make blocking calls.

### 2.3 Migration-Based Load Balancing

The Charm++ runtime framework includes an automatic run-time load balancer, which can monitor the performance of your parallel program. If needed, the load balancer can “migrate” threads from heavily-loaded processors to more lightly-loaded processors, improving the load balance and speeding up the program. For this to be useful, you need to pass the link-time argument `-balancer B` to set the load balancing algorithm, and the run-time argument `+vp N` (use N virtual processors) to set the number of threads. The ideal number of threads per processor depends on the problem, but we’ve found five to a hundred threads per processor to be a useful range.

When a thread migrates, all its data must be brought with it. “Stack data”, such as variables declared locally in a subroutine, will be brought along with the thread automatically. Global data, as described in Section 2.1, is never brought with the thread and should generally be avoided.

“Heap data” in C is structures and arrays allocated using `malloc` or `new`; in Fortran, heap data is TYPES or arrays allocated using `ALLOCATE`. To bring heap data along with a migrating thread, you have two choices: write a pup routine or use isomalloc. Pup routines are described in Section 3.1.

Isomalloc is a special mode which controls the allocation of heap data. You enable isomalloc allocation using the link-time flag “-memory isomalloc”. With isomalloc, migration is completely transparent—all
your allocated data is automatically brought to the new processor. The data will be unpacked at the same
location (the same virtual addresses) as it was stored originally; so even cross-linked data structures that
contain pointers still work properly.

The limitations of isomalloc are:

- Wasted memory. Isomalloc uses a special interface\(^1\) to acquire memory, and the finest granularity that
can be acquired is one page, typically 4KB. This means if you allocate a 2-entry array, isomalloc will
waste an entire 4KB page. We should eventually be able to reduce this overhead for small allocations.

- Limited space on 32-bit machines. Machines where pointers are 32 bits long can address just 4GB \((2^{32})
bytes\) of virtual address space. Additionally, the operating system and conventional heap already use
a significant amount of this space; so the total virtual address space available is typically under 1GB.
With isomalloc, all processors share this space, so with just 20 processors the amount of memory per
processor is limited to under 50MB! This is an inherent limitation of 32-bit machines; to run on more
than a few processors you must use 64-bit machines or avoid isomalloc.

3 Advanced TCharm Programming

The preceding features are enough to write simple programs that use TCharm-based frameworks. These
more advanced techniques provide the user with additional capabilities or flexibility.

3.1 Writing a Pup Routine

The runtime system can automatically move your thread stack to the new processor, but unless you use
isomalloc, you must write a pup routine to move any global or heap-allocated data to the new processor. A
pup (Pack/UnPack) routine can perform both packing (converting your data into a network message) and
unpacking (converting the message back into your data). A pup routine is passed a pointer to your data
block and a special handle called a “pupper”, which contains the network message.

In a pup routine, you pass all your heap data to routines named `pup_type` or `fpup_type`, where `type`
is either a basic type (such as int, char, float, or double) or an array type (as before, but with a “s”
suffix). Depending on the direction of packing, the pupper will either read from or write to the values you
pass—normally, you shouldn’t even know which. The only time you need to know the direction is when
you are leaving a processor, or just arriving. Correspondingly, the pupper passed to you may be deleting
(indicating that you are leaving the processor, and should delete your heap storage after packing), unpacking
(indicating you’ve just arrived on a processor, and should allocate your heap storage before unpacking), or
neither (indicating the system is merely sizing a buffer, or checkpointing your values).

pup functions are much easier to write than explain— a simple C heap block and the corresponding pup
function is:

```c
typedef struct
{
  int n1; /*Length of first array below*/
  int n2; /*Length of second array below*/
  double *arr1; /*Some doubles, allocated on the heap*/
  int *arr2; /*Some ints, allocated on the heap*/
} my_block;

void pup_my_block(pup_er p, my_block *m)
{
  if (pup_isUnpacking(p)) { /*Arriving on new processor*/
    m->arr1=malloc(m->n1*sizeof(double));
    m->arr2=malloc(m->n2*sizeof(int));
  }
}
```

\(^1\) The interface used is `mmap`. 
pup_doubles(p,m->arr1,m->n1);
pup_ints(p,m->arr2,m->n2);
if (pup_isDeleting(p)) { /*Leaving old processor*/
    free(m->arr1);
    free(m->arr2);
}
}

This single pup function can be used to copy the my_block data into a message buffer and free the old heap storage (deleting pupper); allocate storage on the new processor and copy the message data back (unpacking pupper); or save the heap data for debugging or checkpointing.

A Fortran block TYPE and corresponding pup routine is as follows:

MODULE my_block_mod
    TYPE my_block
        INTEGER :: n1,n2x,n2y
        DOUBLE PRECISION, ALLOCATABLE, DIMENSION(:) :: arr1
        INTEGER, ALLOCATABLE, DIMENSION(:, :) :: arr2
    END TYPE
END MODULE

SUBROUTINE pup_my_block(p,m)
    IMPLICIT NONE
    USE my_block_mod
    USE pupmod
    INTEGER :: p
    TYPE(my_block) :: m
    IF (fpup_isUnpacking(p)) THEN
        ALLOCATE(m%arr1(m%n1))
        ALLOCATE(m%arr2(m%n2x,m%n2y))
    END IF
    call fpup_doubles(p,m%arr1,m%n1)
    call fpup_ints(p,m%arr2,m%n2x*m%n2y)
    IF (fpup_isDeleting(p)) THEN
        DEALLOCATE(m%arr1)
        DEALLOCATE(m%arr2)
    END IF
END SUBROUTINE

You indicate to TCHARM that you want a pup routine called using the routine below. An arbitrary number of blocks can be registered in this fashion.

void TCHARM_Register(void *block, TCharmPupFn pup_fn)
SUBROUTINE TCHARM_Register(block,pup_fn)
    TYPE(varies), POINTER :: block
    SUBROUTINE :: pup_fn

Associate the given data block and pup function. Can only be called from the parallel context. For the declarations above, you call TCHARM_Register as:

    /*In C/C++ driver() function*/
    my_block m;
    TCHARM_Register(m,(TCharmPupFn)pup_my_block);
!- In Fortran driver subroutine
use my_block_mod
interface
  subroutine pup_my_block(p,m)
  use my_block_mod
  INTEGER :: p
  TYPE(my_block) :: m
  end subroutine
end interface
TYPE(my_block), TARGET :: m
call TCHARM_Register(m,pup_my_block)

Note that the data block must be allocated on the stack. Also, in Fortran, the "TARGET" attribute
must be used on the block (as above) or else the compiler may not update values during a migration, because
it believes only it can access the block.

void TCHARM_Migrate()
subroutine TCHARM_Migrate()

Informs the load balancing system that you are ready to be migrated, if needed. If the system decides to
migrate you, the pup function passed to TCHARM_Register will first be called with a sizing pupper, then a
packing, deleting pupper. Your stack and pupped data will then be sent to the destination machine, where
your pup function will be called with an unpacking pupper. TCHARM_Migrate will then return. Can only
be called from in the parallel context.

3.2 Readonly Global Variables

You can also use a pup routine to set up initial values for global variables on all processors. This pup routine
is called with only a pup handle, just after the serial setup routine, and just before any parallel context
routines start. The pup routine is never called with a deleting pup handle, so you need not handle that case.
A C example is:

int g_arr[17];
double g_f;
int g_n; /*Length of array below*/
float *g_allocated; /*heap-allocated array*/

void pup_my_globals(pup_er p)
{
  pup_ints(p,g_arr,17);
pup_double(p,&g_f);
pup_int(p,&g_n);
  if (pup_isUnpacking(p)) { /*Arriving on new processor*/
    g_allocated=malloc(g_n*sizeof(float));
  }
pup_floats(p,g_allocated,g_n);
}

A fortran example is:

MODULE my_globals_mod
  INTEGER :: g_arr(17)
  DOUBLE PRECISION :: g_f
  INTEGER :: g_n
  SINGLE PRECISION, ALLOCATABLE :: g_allocated(:)

8
You register your global variable pup routine using the method below. Multiple pup routines can be registered the same way.

```c
void TCHARM_ReadonlyGlobals(TCharmPupGlobalFn pup_fn)
SUBROUTINE TCHARM_Readonly_globals(pup_fn)
  SUBROUTINE :: pup_fn

4 Combining Frameworks

This section describes how to combine multiple frameworks in a single application. You might want to do this, for example, to use AMPI communication inside a finite element method solver.

You specify how you want the frameworks to be combined by writing a special setup routine that runs when the program starts. The setup routine must be named TCHARM_User_setup. If you declare a user setup routine, the standard framework setup routines (such as the FEM framework’s init routine) are bypassed, and you do all the setup in the user setup routine.

The setup routine creates a set of threads and then attaches frameworks to the threads. Several different frameworks can be attached to one thread set, and there can be several sets of threads; however, the most frameworks cannot be attached more than once to single set of threads. That is, a single thread cannot have two attached AMPI frameworks, since the MPI_COMM_WORLD for such a thread would be indeterminate.

```c
Create a new set of TCHARM threads of the given size. The threads will execute the given function, which is normally your user code. You should call TCHARM_Get_num_chunks() to get the number of threads from the command line. This routine can only be called from your TCHARM_User_setup routine.

You then attach frameworks to the new threads. The order in which frameworks are attached is irrelevant, but attach commands always apply to the current set of threads.

To attach a chare array to the TCHARM array, use:

```c
CkArrayOptions TCHARM_Attach_start(CkArrayID *retCkArrayArray,int *retNumElts)
This function returns a CkArrayOptions object that will bind your chare array to the TCHARM array, in addition to returning the TCHARM array proxy and number of elements by reference. If you are using frameworks like AMPI, they will automatically attach themselves to the TCHARM array in their initialization routines.

9
5 Command-line Options

The complete set of link-time arguments relevant to TCharm is:

- **-memory isomalloc** Enable memory allocation that will automatically migrate with the thread, as described in Section 2.3.

- **-balancer B** Enable this load balancing strategy. The current set of balancers B includes RefineLB (make only small changes each time), MetisLB (remap threads using graph partitioning library), HeapCentLB (remap threads using a greedy algorithm), and RandCentLB (remap threads to random processors). You can only have one balancer.

- **-module F** Link in this framework. The current set of frameworks F includes ampi, collide, fem, mblock, and netfem. You can link in multiple frameworks.

The complete set of command-line arguments relevant to TCharm is:

- **+p N** Run on N physical processors.

- **+vp N** Create N “virtual processors”, or threads. This is the value returned by TCharmGetNumChunks.

- **++debug** Start each program in a debugger window. See Charm++ Installation and Usage Manual for details.

- **+tcharm_stacksize N** Create N-byte thread stacks. This value can be overridden using TCharmSetStackSize().

- **+tcharm_nomig** Disable thread migration. This can help determine whether a problem you encounter is caused by our migration framework.

- **+tcharm_nothread** Disable threads entirely. This can help determine whether a problem you encounter is caused by our threading framework. This generally only works properly when using only one thread.

- **+tcharm_trace F** Trace all calls made to the framework F. This can help to understand a complex program. This feature is not available if Charm++ was compiled with CMK_OPTIMIZE.

6 Writing a library using TCharm

Until now, things were presented from the perspective of a user—one who writes a program for a library written on TCharm. This section gives an overview of how to go about writing a library in Charm++ that uses TCharm.

- Compared to using plain MPI, TCharm provides the ability to access all of Charm++, including arrays and groups.

- Compared to using plain Charm++, using TCharm with your library automatically provides your users with a clean C/F90 API (described in the preceding chapters) for basic thread memory management, I/O, and migration. It also allows you to use a convenient "thread->suspend()" and "thread->resume()" API for blocking a thread, and works properly with the load balancer, unlike CthSuspend/CthAwaken.

The overall scheme for writing a TCharm-based library "Foo" is:

1. You must provide a FOO_Init routine that creates anything you’ll need, which normally includes a Chare Array of your own objects. The user will call your FOO_Init routine from their main work routine; and normally FOO_Init routines are collective.
2. In your FOO_Init routine, create your array bound it to the running TCharm threads, by creating it using the CkArrayOptions returned by TCHARM_Attach_start. Be sure to only create the array once, by checking if you’re the master before creating the array.

One simple way to make the non-master threads block until the corresponding local array element is created is to use TCharm semaphores. These are simply a one-pointer slot you can assign using TCharm::semaPut and read with TCharm::semaGet. They’re useful in this context because a TCharm::semaGet blocks if a local TCharm::semaGet hasn’t yet executed.

```c
//This is either called by FooFallbackSetuo mentioned above, or by the user
//directly from TCHARM_User_setup (for multi-module programs)
void FOO_Init(void)
{
    if (TCHARM_Element()==0) {
        CkArrayID threadsAID; int nchunks;
        CkArrayOptions opts=TCHARM_Attach_start(&threadsAID,&nchunks);

        //actually create your library array here (FooChunk in this case)
        CkArrayID aid = CProxy_FooChunk::ckNew(opt);
    }
    FooChunk *arr=(FooChunk *)TCharm::semaGet(FOO_TCHARM_SEMAID);
}
```

3. Depending on your library API, you may have to set up a thread-private variable(Ctv) to point to your library object. This is needed to regain context when you are called by the user. A better design is to avoid the Ctv, and instead hand the user an opaque handle that includes your array proxy.

```c
//_fooptr is the Ctv that points to the current chunk FooChunk and is only valid in
//routines called from fooDriver()
CtvStaticDeclare(FooChunk *, _fooptr);

/* The following routine is listed as an initcall in the .ci file */
/*initcall*/ void fooNodeInit(void)
{
    CtvInitialize(FooChunk*, _fooptr);
}
```

4. Define the array used by the library

```c
class FooChunk: public TCharmClient1D {
    CProxy_FooChunk thisProxy;

protected:
    //called by TCharmClient1D when thread changes
    virtual void setupThreadPrivate(CthThread forThread)
    {
        CtvAccessOther(forThread, _fooptr) = this;
    }

    FooChunk(CkArrayID aid):TCharmClient1D(aid)
    {
        thisProxy = this;
        tCharmClientInit();
        TCharm::semaPut(FOO_TCHARM_SEMAID,this);
        //add any other initialization here
```
virtual void pup(PUP::er &p) {
    TCharmClient1D::pup(p);
    // usual pup calls
}

// ... any other calls you need...
int doCommunicate(...);
void recvReply(someReplyMsg *m);
..........  

5. Block a thread for communication using thread->suspend and thread->resume

int FooChunk::doCommunicate(...) {
    replyGoesHere = NULL;
    thisProxy[destChunk].sendRequest(...);
    thread->suspend(); // wait for reply to come back
    return replyGoesHere->data;
}

void FooChunk::recvReply(someReplyMsg *m) {
    if(replyGoesHere!=NULL) CkAbort("FooChunk: unexpected reply n");
    replyGoesHere = m;
    thread->resume(); // Got the reply -- start client again
}

6. Add API calls. This is how user code running in the thread interacts with the newly created library.
Calls to TCHARM_API_TRACE macro must be added to the start of every user-callable method. In addition to tracing, these disable isomalloc allocation.

The charm-api.h macros CDECL, FDECL and FTN_NAME should be used to provide both C and
FORTRAN versions of each API call. You should use the "MPI capitalization standard", where
the library name is all caps, followed by a capitalized first word, with all subsequent words lowercase,
separated by underscores. This capitalization system is consistent, and works well with case-insensitive
languages like Fortran.

Fortran parameter passing is a bit of an art, but basically for simple types like int (INTEGER in
fortran), float (SINGLE PRECISION or REAL*4), and double (DOUBLE PRECISION or REAL*8),
things work well. Single parameters are always passed via pointer in Fortran, as are arrays. Even
though Fortran indexes arrays based at 1, it will pass you a pointer to the first element, so you can use
the regular C indexing. The only time Fortran indexing need be considered is when the user passes
you an index—the int index will need to be decremented before use, or incremented before a return.

CDECL void FOO_Communicate(int x, double y, int * arr) {
    TCHARM_API_TRACE("FOO_Communicate", "foo"); // 2nd parameter is the name of the library
    FooChunk *f = CtvAccess(<foooptr>);
    f->doCommunicate(x, y, arr);
}


// In fortran, everything is passed via pointers
FDECL void FTN_NAME(FOO_COMMUNICATE, foo_communicate)
(int *x, double *y, int *arr)
{
    TCHARM_API_TRACE("FOO_COMMUNICATE", "foo");
    FooChunk *f = CtvAccess(_fooptr);
    f->doCommunicate(*x, *y, arr);
}