# Change Log

|  |  |
| --- | --- |
| v0 | Initial Version by Terry Wilmarth |
| v1 | Minor updates |
| v2 | API changes |
| v3 | Example of iterator usage added |
| v4 | Minor fixes |

# Task Affinity API

The following is the proposed API for the new task affinity clause to be added to the task construct in the upcoming OpenMP 5.0 specification.

#pragma omp task ... affinity(list[:iterators-definition]) …

This API should be called in the compiler code gen after the task has been allocated via the \_\_kmpc\_omp\_task\_alloc call. The affinity information is registered in the task thunk returned by that call to \_\_kmpc\_omp\_task\_alloc using a call to \_\_kmpc\_omp\_reg\_task\_with\_affinity. The number of items in the list given in the affinity clause should be passed into the API in the naffins parameter, and the list items themselves are passed into the API via the affin\_list parameter. Once the affinity information has been registered, the task can be deferred for execution by calling \_\_kmpc\_omp\_task() or \_\_kmp\_omp\_task\_with\_deps as normally done by the compiler for tasks and tasks with dependencies, respectively.

Each item in the affin\_list is of type kmp\_task\_affinity\_info\_t, shown below. There may be some number of flags associated with each item, the quantity and names of these flags can be decided later, but for the moment we have set aside sizeof(kmp\_int32) to hold the flags.

typedef struct kmp\_task\_affinity\_info {

kmp\_intptr\_t base\_addr;

size\_t len;

union {

struct {

bool flag1 : 1;

bool flag2 : 1;

} s;

kmp\_int32 pad\_flags;

} flags;

} kmp\_task\_affinity\_info\_t;

/\*!

@ingroup TASKING

@param loc\_ref location of the original task directive

@param gtid Global Thread ID of encountering thread

@param new\_task task thunk allocated by \_\_kmpc\_omp\_task\_alloc() for the ''new

task''

@param naffins Number of affinity items

@param affin\_list List of affinity items

@return Returns non-zero if registering affinity information was not successful. Returns 0 if registration was successful

This entry registers the affinity information attached to a task with the task thunk structure kmp\_taskdata\_t.

\*/

kmp\_int32 \_\_kmpc\_omp\_reg\_task\_with\_affinity(ident\_t \*loc\_ref, kmp\_int32 gtid,

kmp\_task\_t \*new\_task, kmp\_int32 naffins,

kmp\_task\_affinity\_info\_t \*affin\_list);

## Code Generation Example

For the following example:

int i = 32;

double j = 3.14;

#pragma omp task affinity(i, j)

The following code should be generated:

t = \_\_kmpc\_omp\_task\_alloc(loc\_struct, gtid, <flags>,

<sizeof\_kmp\_task\_t>, <sizeof\_shareds>, task\_entry);

int naff = 2;

kmp\_task\_affinity\_info\_t affin\_list[naff];

affin\_list[0].len = sizeof(i);

affin\_list[0].base\_addr = &i;

affin\_list[0].flags.pad\_flags = 0;

affin\_list[1].len = sizeof(j);

affin\_list[1].base\_addr = &j;

affin\_list[1].flags.pad\_flags = 0;

\_\_kmpc\_omp\_reg\_task\_with\_affinity(loc\_struct, gtid, t,

naff, affin\_list);

\_\_kmpc\_omp\_task(...t...); // \_\_kmp\_omp\_task\_with\_deps(...t...);

For multiple occurrences of the affinity clause, the compiler emits the same code:

int i = 32;

double j = 3.14;

#pragma omp task affinity(i) affinity(j)

The following code is generated:

t = \_\_kmpc\_omp\_task\_alloc(loc\_struct, gtid, <flags>,

<sizeof\_kmp\_task\_t>, <sizeof\_shareds>, task\_entry);

int naff = 2;

kmp\_task\_affinity\_info\_t affin\_list[naff];

affin\_list[0].len = sizeof(i);

affin\_list[0].base\_addr = &i;

affin\_list[0].flags.pad\_flags = 0;

affin\_list[1].len = sizeof(j);

affin\_list[1].base\_addr = &j;

affin\_list[1].flags.pad\_flags = 0;

\_\_kmpc\_omp\_reg\_task\_with\_affinity(loc\_struct, gtid, t,

naff, affin\_list);

\_\_kmpc\_omp\_task(...t...); // \_\_kmp\_omp\_task\_with\_deps(...t...);

For affinity clauses that that contain the iterator clause, the compiler has to generate a loop to iterate and store the affinity information:

int \*\*a = init\_this\_thing();

double deliciouos\_pi = 3.14;

#pragma omp task affinity(deliciouos\_pi) affinity(a[i][j]: i=0:j)

Will become:

t = \_\_kmpc\_omp\_task\_alloc(loc\_struct, gtid, <flags>,

<sizeof\_kmp\_task\_t>, <sizeof\_shareds>, task\_entry);

int naff = 1 + j;

kmp\_task\_affinity\_info\_t affin\_list[naff];

affin\_list[0].len = sizeof(deliciouos\_pi);

affin\_list[0].base\_addr = &deliciouos\_pi;

affin\_list[0].flags.pad\_flags = 0;

for (tmp = 0; tmp < j; tmp++) {

affin\_list[tmp+1].len = sizeof(...); // if possible to determine

affin\_list[tmp+1].base\_addr = &a[i][tmp];

affin\_list[tmp+1].flags.pad\_flags = 0;

}

\_\_kmpc\_omp\_reg\_task\_with\_affinity(loc\_struct, gtid, t,

naff, affin\_list);

\_\_kmpc\_omp\_task(...t...); // \_\_kmp\_omp\_task\_with\_deps(...t...);

## Runtime Implementation Notes

The kmp\_taskdata\_t structure is extended with two new members “kmp\_task\_affinity\_info\_t \*affinity\_info” and a “kmp\_int32 naffin”.

The entry point \_\_kmpc\_omp\_reg\_task\_with\_affinity registers the affinity information by allocation memory to store a copy of the affinity data coming from the caller. This needs to be done, so that the affinity data is guaranteed to live as long as the task structure is needed by the runtime. After the copy has been created, the entry point returns.

The length field of the kmp\_task\_affinity\_info\_t structure is not needed to implement the feature as defined in the OpenMP API specification. Passing it may enable future use and may be used for debugging purposes.