PTK: A Parallel Toolkit Library

Kirsten Allison and Amit Jain

Boise State University

November 3, 2010

The High Performance Computing Market

- The cluster market is growing with the increasing price/performance ratio of commodity computing hardware.
- Cluster software development is not necessarily keeping up.
- There are many specialized software packages for certain applications, including computational chemistry and biology, oceanic and atmospheric modeling, and copious math libraries.
- There are also applications for building clusters, and system administration tools.
- Programs are still being written at the message passing level.

Writing Programs At the Message Passing Level

To send a “chunk” of an array from one node to all the other nodes in the group we do:

```
if (me == sender) {
    for(i = 0; i < size of the group - 1; i++) {
        pack the i’th chunk into a send buffer
        send the data
    }
} else {
    probe for a message and find out how big it is
    allocate memory to receive the message
    receive the message and unpack it
}
```

It would be much simpler to do:

```
ptk_scatter(data, ...);
```

PTK: Parallel Toolkit Library

- The toolkit supports common parallel program design patterns.
  - Data sharing - scatter, gather, all to all, multicast.
  - Workpools - centralized and distributed.
  - Utilities - initialize, exit, filemerge.
- The library builds on PVM and MPI, which are lower level message passing libraries that are widely used to write parallel programs.
- Examples are provided to demonstrate how to use PTK.
- Documentation on the toolkit functions and examples is included.

Prior Research

Nathan Sachs and Jeff McGough: "Hybrid Process Farm/Work Pool Implementation in a Distributed Environment using MPI."

- Presented at the Midwest Instructional Computing Symposium, Duluth, Minnesota, April 2003.
- Part of a project with Sun to develop preliminary versions of their libraries.
- Not available via open source.

Steffen Priebe: "Dynamic Task Generation and Transformation within a Nestable Workpool Skeleton."

- Presented at the European Conference on Parallel Computing (Euro-Par) 2006, Dresden, Germany.
- Skeletons are written in Eden, a parallel version of Haskell.
- C and Fortran are the predominant parallel programming languages. Eden is not mainstream enough to be pertinent to our discussion.

General Issues in Library Development

- Adequate functionality versus ease of use:
  - Adding functionality means adding parameters.
  - The more parameters there are, the harder it is to understand how to use the function, however ...
  - the additional parameters give us more functionality.
- Memory allocation - where should it happen?
  - Wanted the library to do as much for the user as possible.
  - Inconsistent to have the toolkit allocate memory and then expect user to free it.
  - Conclusion was to have the user allocate and deallocate memory wherever possible.

Simplifying Initialization and Exit

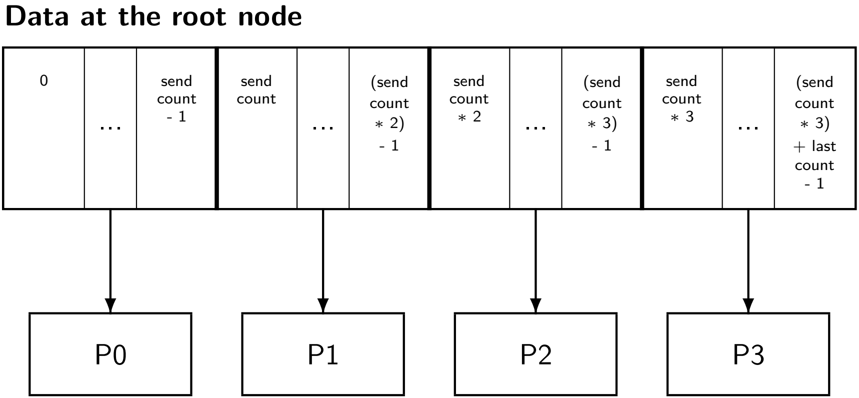
- Initialization
  - Every PVM and MPI program starts by doing the same things.
  - Variables are filled in, such as group size, group rank, task IDs (PVM), etc.
  - Instead of three or four PVM or MPI calls, this is now one function.
- Exit
  - Same problem as initialization.
  - Exit is now also one function, and ensures that the necessary things are done before a program exits.

Data Sharing Design Patterns

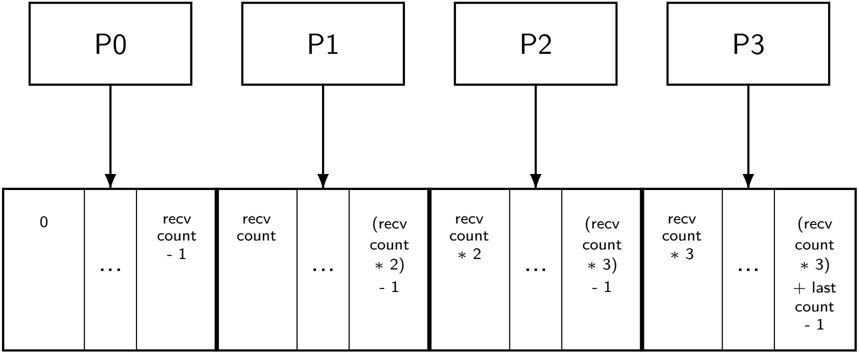
The PTK data sharing functions do not have the limitations that the MPI functions do:

- Arrays do not need to be evenly divisible by the number of processes in the group.
- PTK supports data sharing of two-dimensional arrays.
- Patterns supported include:
  - Scatter
  - Gather
  - All to all
  - Multicast

Scatter with a group size of four



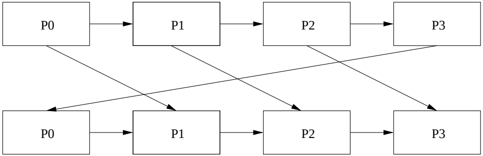
Gather



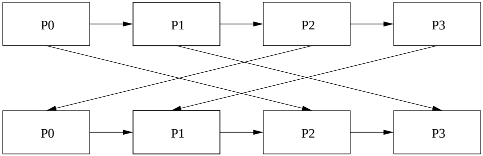
Data at the root node

All to all

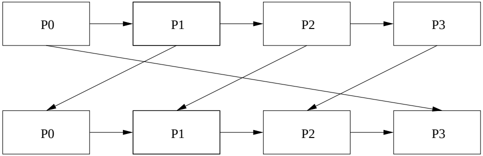
First iteration of all to all



Second iteration of all to all



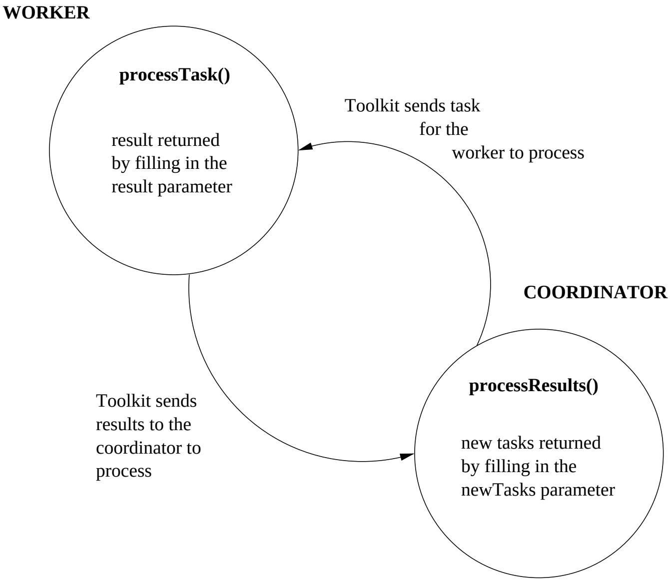
Third iteration of all to all



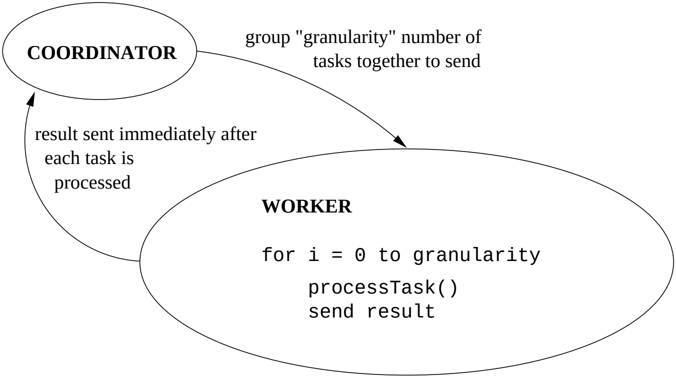
Workpools

- Centralized may be used when:
  - a task can be processed by any node,
  - the worker nodes can store all of the data needed to process tasks,
  - the coordinator can store all of the data needed to process results.
- Distributed may be used when:
  - nodes can be responsible for a certain set of tasks,
  - the memory required to process tasks can not be stored at one node.

Processing tasks and results using the centralized workpool



Using granularity in the centralized workpool



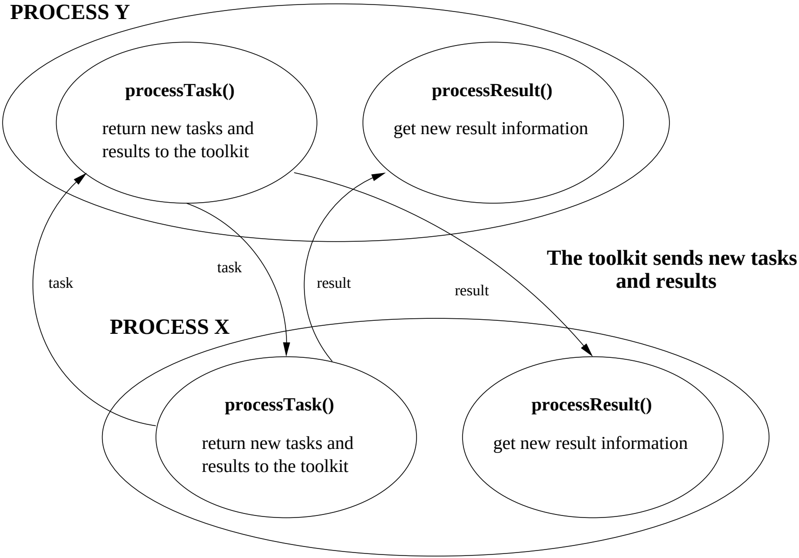
Example processTask() function - Centralized

```
int sqr(void *dataToProcess,
        void **ptkResult,
        int *returnSize)
{
    long int *intData = (long int *)dataToProcess;
    long int number = *intData * *intData;
    memcpy(*ptkResult, &number, sizeof(long int));
    *returnSize = sizeof(long int);
    return 0;
}
```

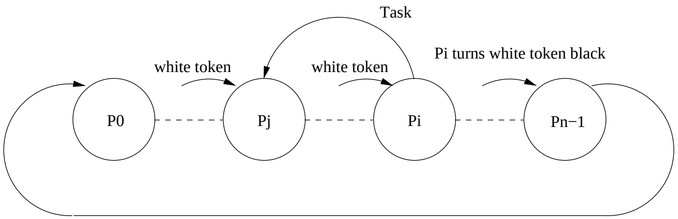
Example `processResult()` function - Centralized

```
int processResult(void *results,
                  void **ptkNewTasks,
                  int *numNewTasks)
{
    long int *intResults = (long int *)results;
    sum += *intResults;
    *ptkNewTasks = NULL;
    *numNewTasks = 0;
    return 0;
}
```

Processing tasks and results using the distributed workpool



Dual-pass token ring termination algorithm

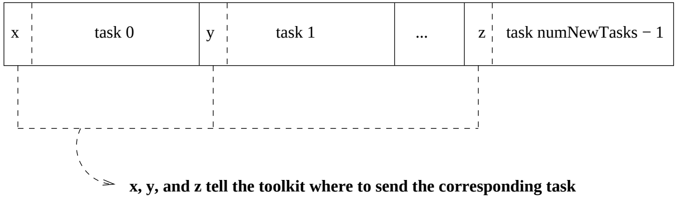


Example processTask() function - Distributed

```
void *processTask(void *task, int tasksToProcess,
                    void **ptkNewTasks, int *numNewTasks,
                    void **ptkResults, int *numResults) {

    for (i = 0; i < tasksToProcess; i++) {
        processTask;
        if (new task generated) {
            increment numNewTasks;
            copy new task into ptkNewTasks;
        }
    }
    copy results into pktResults;
    set numResults value;
}
```

Structure of a task in the distributed workpool



Example processResult() function - Distributed

```
void *processResult(void *result) {

    for (i = 0; i < length of result array; i++) {
        process ith result;
    }
}
```

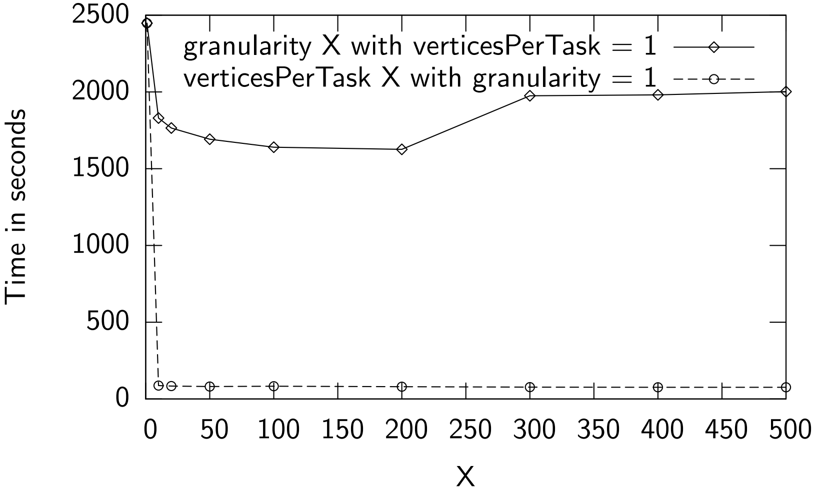
Testing coverage of toolkit functions

Toolkit function	Example/test program
ptk_alltoall1d	allToAll1d
ptk_alltoall2d	bucketSortWithAlltoAll2d
ptk_central_workpool	shortestPathsCentral shortestPathsCentralMoreEfficient sumOfSquares
ptk_distributed_workpool	shortestPathsDistributed shortestPathsDistributedMoreEfficient
ptk_exit	all
ptk_filemerge	filemerge
ptk_gather1d	gather1d
ptk_gather2d	gather2d
ptk_init	all
ptk_mcast	shortestPathsCentral shortestPathsCentralWithGranularity
ptk_scatter1d	shortestPathsDistributed
ptk_scatter2d	bucketSortWithScatter2d

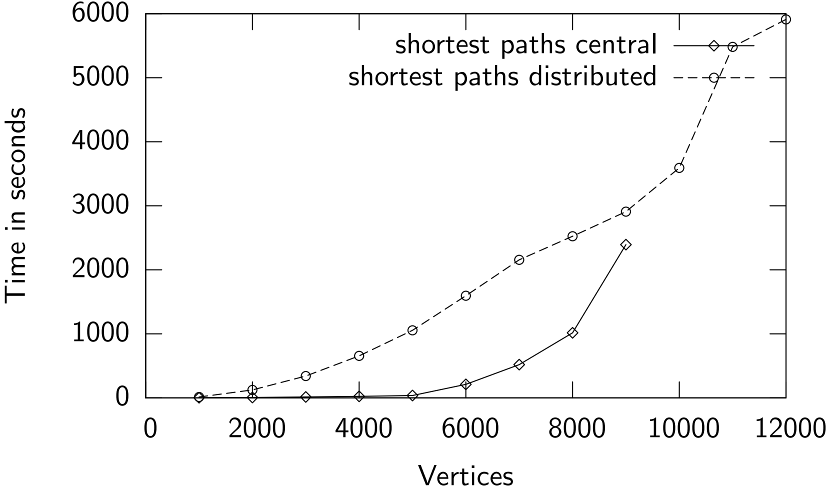
Shortest Paths Examples

- The shortest paths programs are the most significant examples of how to use the workpools.
- They use Moore's algorithm for finding shortest paths in a directed graph with positive edge weights.
- Vertices to investigate are kept in a queue. For each vertex j in the queue do the following:
  1. Find the distance to vertex j through vertex i and compare with the current minimum distance to vertex j.
  2. Change the minimum distance if the distance through vertex i is shorter.
  3. If a new distance is found for vertex j, insert it into the queue.

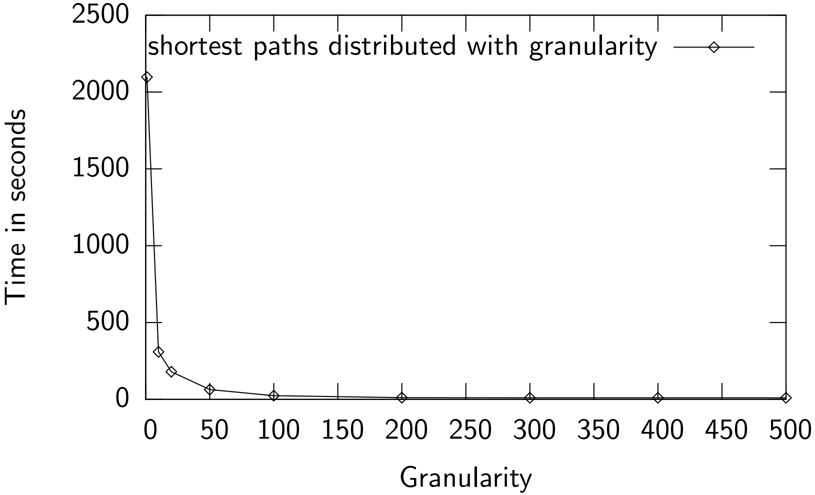
Benchmarking: Shortest paths central



Benchmarking:
Shortest paths central versus shortest paths distributed



Benchmarking: Shortest paths distributed



Summary

- PTK contains functions for data sharing that go beyond what is available in PVM and MPI.
- PTK contains workpools that are not available via open source. The workpools provide functionality that would require a significant amount of time to create from scratch.
- The functions in PTK are fully tested, and benchmarking numbers are available.
- PTK provides the user with examples of how to use the functions, along with user documentation.

Future Directions

- Change the task list in the centralized workpool to a priority queue.
  - this may slow things down - needs to be benchmarked
  - adds an extra parameter
- Multi-threading - implement separate computation and communication threads in the distributed workpool.
- Create a C++ version of the library.