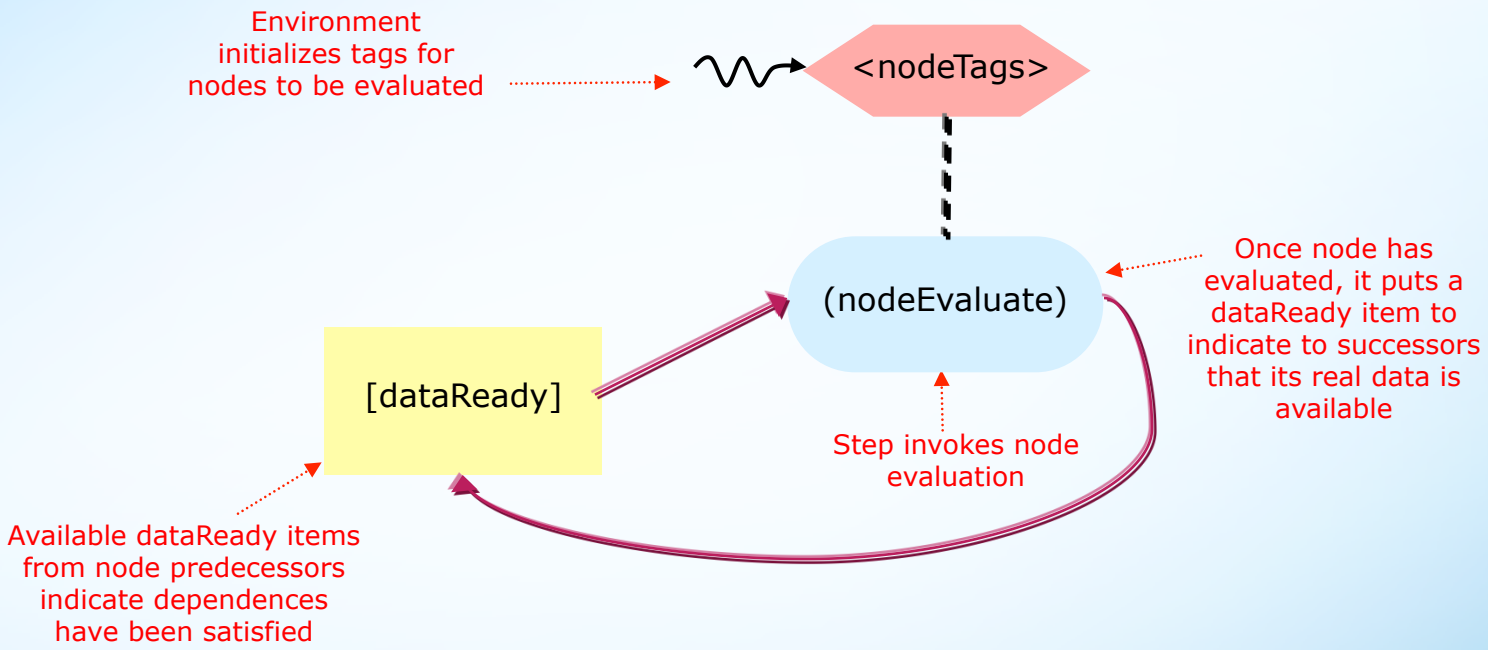
Optimizing CnC's Evaluation of DWA's Dependency Graphs

Mark Hampton, Intel SSG-DPD-TPI

The Third Annual Concurrent Collections Workshop
September 7, 2011

As described previously, CnC maps in a straightforward manner to the dependency graphs used by DWA



- The DWA program structure provides a form of dynamic single assignment:
  - DWA nodes in the graph currently store separate copies of input and output data
  - We only invoke CnC for a single frame at a time
- Combining the above approach with CnC's depends() tuning method (developed due to the DWA project) provides a simple way to evaluate DAGs with no requeueing
- However, we had to address a key question: was a straightforward solution sufficient to improve performance for DWA's workloads?

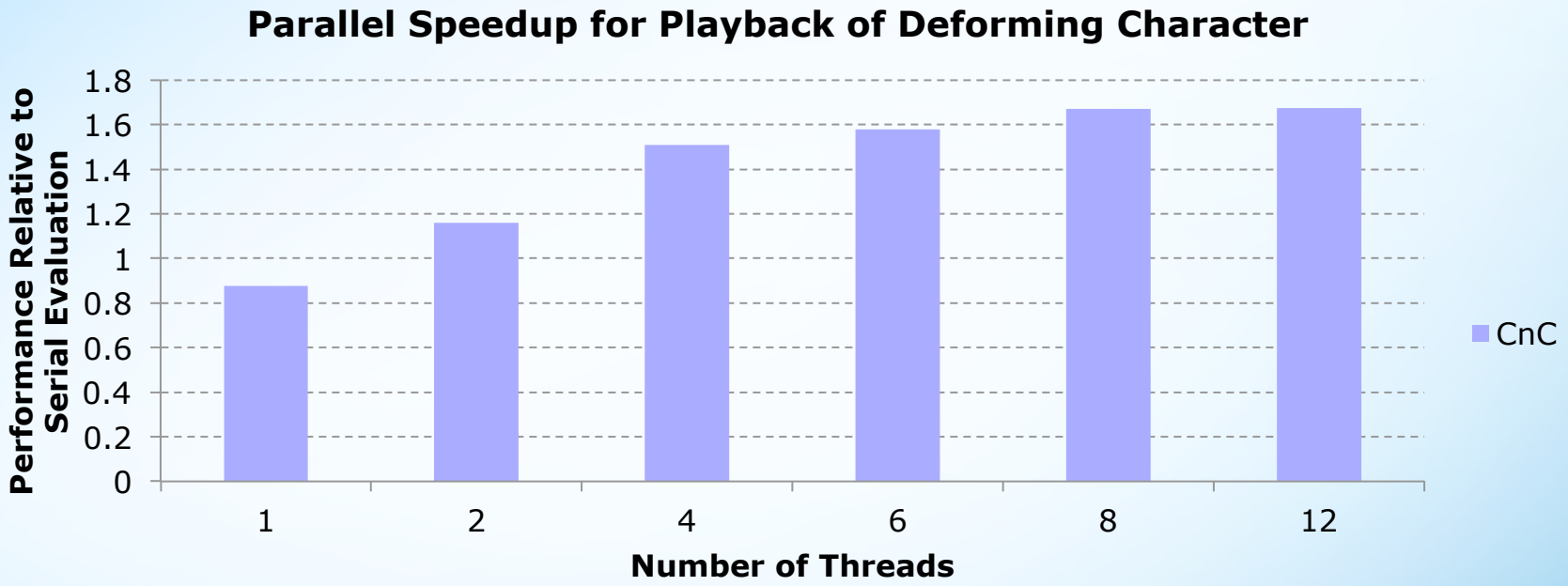
Here is an outline of what we will be covering

- Performance results from a simple implementation
- Reduction of CnC runtime system overhead
  - Optimizations "inside" of the CnC model
  - Optimizations "outside" of the CnC model
- Additional optimizations to consider

Martin discussed the different characteristics of the motion and deformation systems

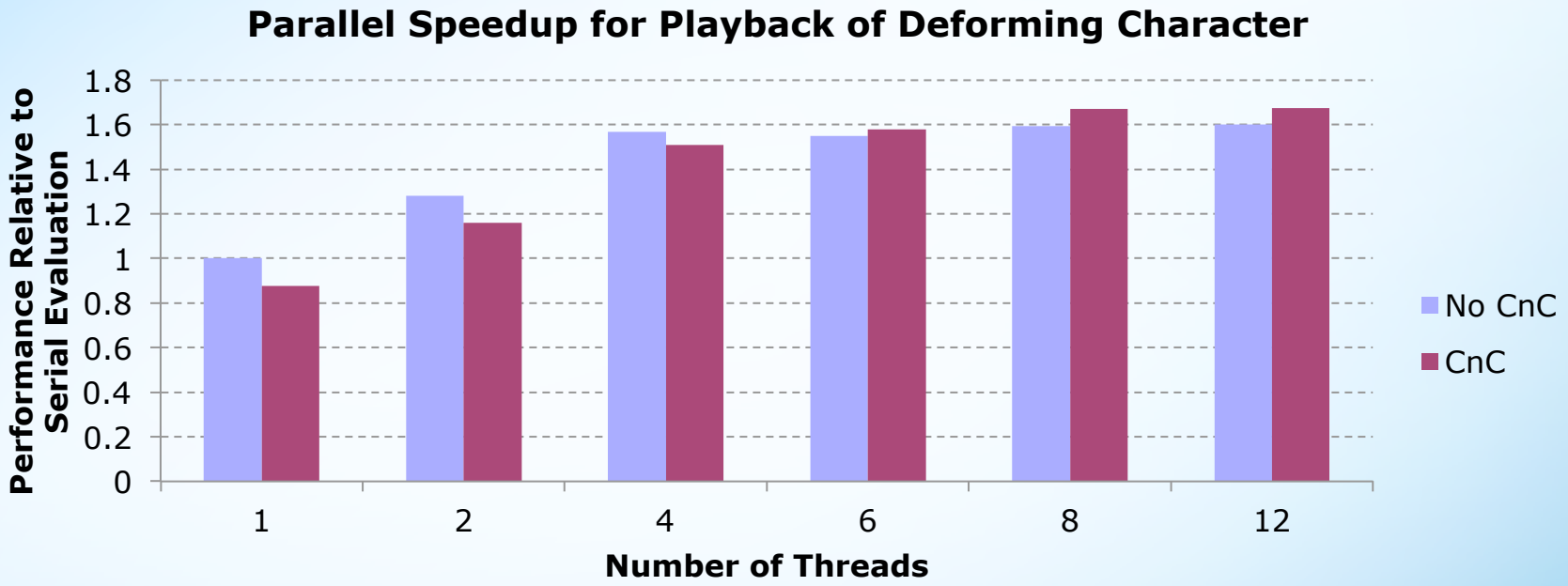
- The motion system contains 100s to 1000s of very lightweight nodes
  - For all results I show in this talk, many of the nodes in the motion system have sub-microsecond runtimes
  - Nodes with runtimes less than a microsecond are typically more lightweight than the recommended threshold for a TBB task (and thus for a CnC step instance)
  - As a result, the motion system magnifies any overhead from CnC; but that system is also the main source of graph-level parallelism in a single-character workload
- The deformation system contains 10s of much heavier nodes
  - The most heavyweight nodes are part of a linear chain, and thus can't be executed in parallel
  - Much of the expected performance gain in the deformation system comes from internal node parallelism (parallel_for constructs)
- The motion system feeds into the deformation system in a full character graph

At first glance, straightforward CnC appears to achieve some small level of speedup



- All results in this talk were obtained on a 12-core Westmere (2 sockets, 6 cores per socket) with Hyper-Threading disabled
- A deforming character contains both the motion and deformation systems
- Performance isn't great, but at least there is a speedup starting with 2 threads

However, when factoring out internal node parallelism, the results don't look too good



- The "No CnC" bars correspond to disabling graph-level parallelism—the only speedup comes from parallel_for constructs inside heavyweight nodes in the deformation system
- Using CnC actually results in worse performance than turning CnC off for up to 6 threads

What are some potential areas of optimization that we can explore in CnC?

- Reducing runtime system overhead
  - Overhead of creating and launching step instances
  - Overhead of managing dependencies
- Step scheduling
  - Steps on the critical path should have highest priority
  - Producers/consumers should run consecutively if possible to exploit locality
- Step affinity to facilitate data sharing
  - Affinity to other steps: group steps together
  - Affinity to a processor: make a step run on a specific core
- Memory reuse is a typical problem for CnC, but…
  - …this doesn't apply in our scenario because we use dummy CnC items and only evaluate a single frame at a time

Here is an outline of what we will be covering

- Performance results from a simple implementation
- Reduction of CnC runtime system overhead
  - Optimizations "inside" of the CnC model
  - Optimizations "outside" of the CnC model
- Additional optimizations to consider

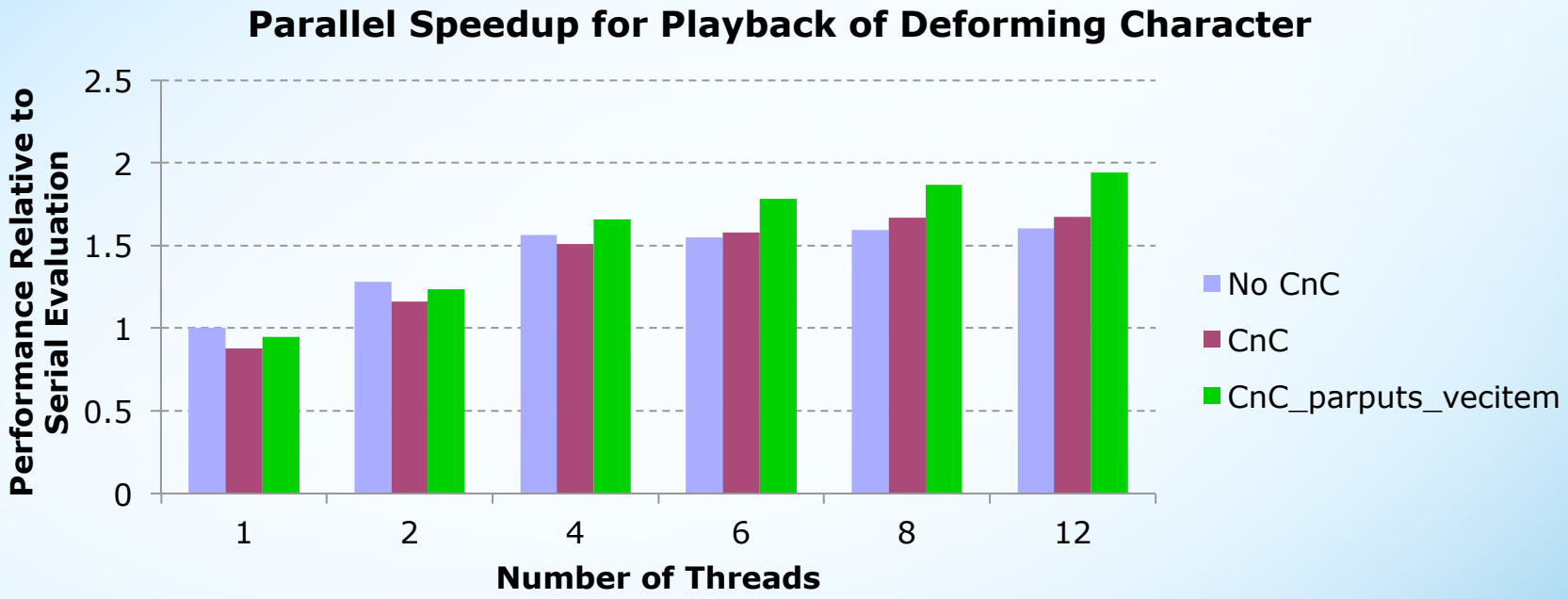
One area of runtime system overhead is the process of putting tags

- In a straightforward implementation, the main thread puts all of the initial tags
- As a result, the real work does not begin until *either*…
  - …the main thread finishes putting all of the tags and starts accessing its work queue, *or*…
  - …another thread steals work from the main thread's queue
- If you are putting thousands of tags from the environment, you can waste a significant amount of time
- By parallelizing tag-puts (using a TBB or CnC parallel_for construct), you can improve your efficiency
  - Tags are put more quickly, allowing the real work to begin sooner
  - Threads which steal the work of putting tags will also end up creating step instances (for node evaluations) in their own work queues, rather than having to steal them from the main thread

Another area of overhead is the use of item collections (even though they don't hold real data)

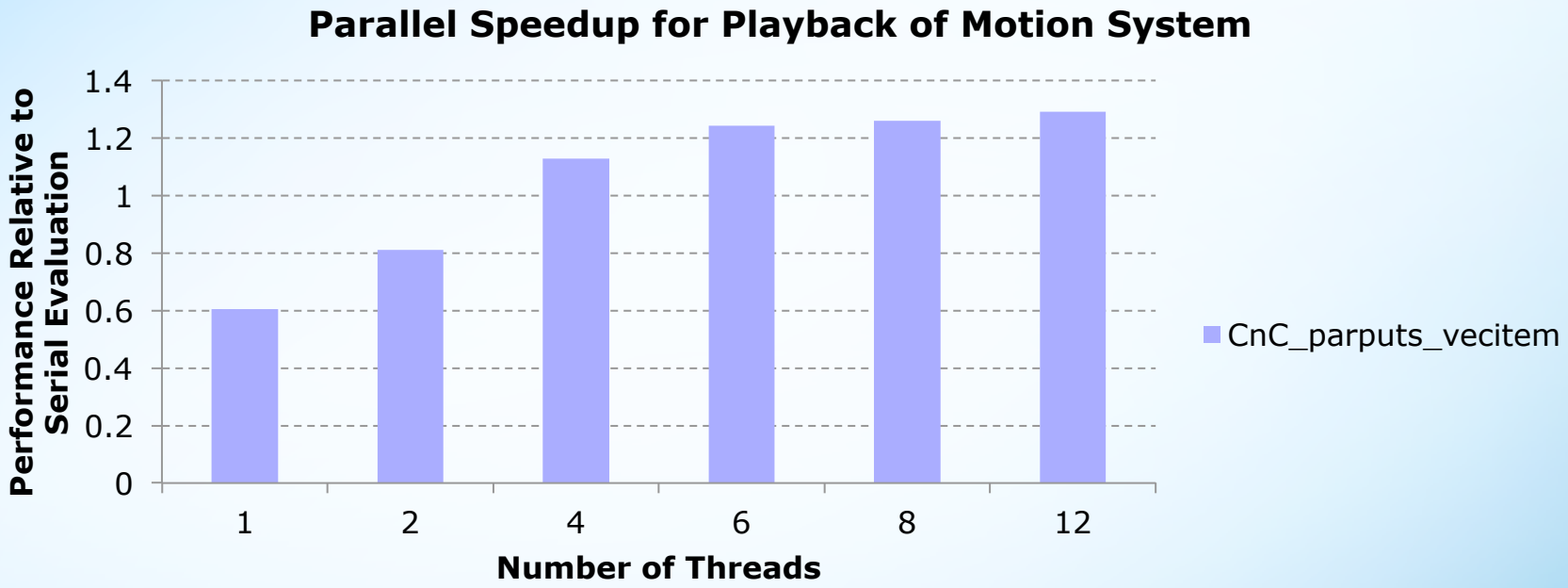
- Each usage of an item instance requires a hash table access
- Because our tags are integers (which index a vector of node pointers), using a hash table is unnecessarily inefficient
- Motivated by the DWA project, Frank Schlimbach developed vector-based item collections in CnC
  - Must be used with integer tags
  - The user must specify the maximum possible tag-value

The two previous optimizations help to improve performance a bit



- Most of the benefit comes from parallelizing the tag-puts (only about a couple percent improvement comes from vector-based item collections)
- The use of CnC (with optimizations) now demonstrates a slight performance edge at 4 threads, although that is still not ideal (we want a performance improvement starting with 2 threads)

What about just the motion system by itself?



- The motion system is the primary source of graph-level parallelism, so it allows us to better see how much benefit CnC is really providing
- Without the optimizations previously described, CnC is always worse than serial evaluation (>30% performance degradation)
- With optimizations, we see a slight performance edge over serial evaluation at 4 threads

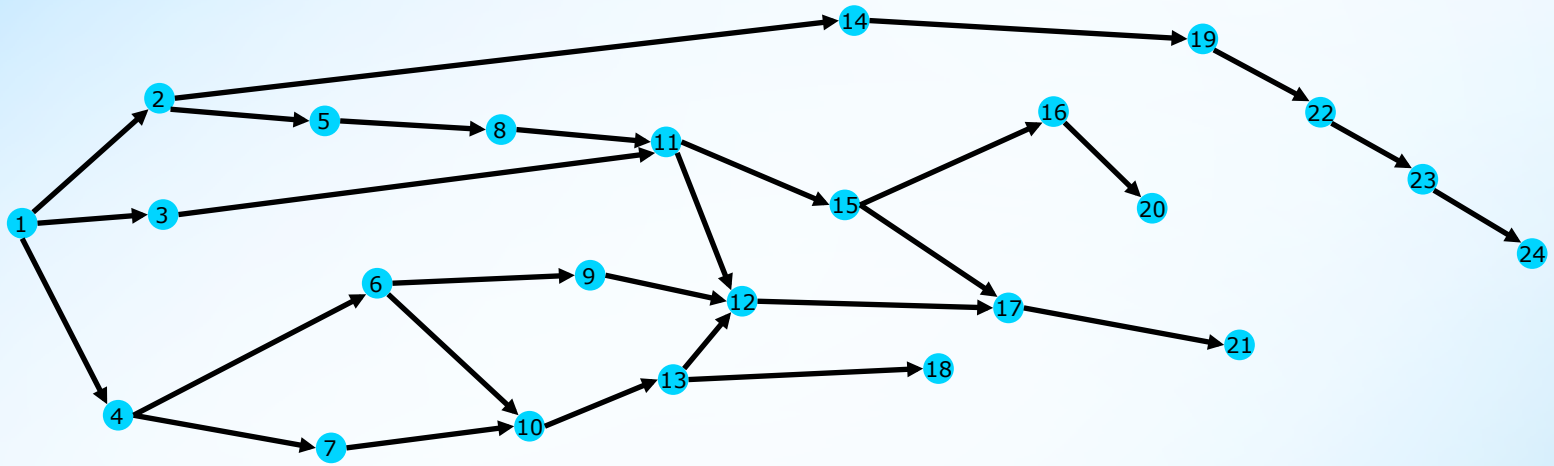
Here is an outline of what we will be covering

- Performance results from a simple implementation
- Reduction of CnC runtime system overhead
  - Optimizations "inside" of the CnC model
  - Optimizations "outside" of the CnC model
- Additional optimizations to consider

One problem is that even though CnC's depends() method avoids requeueing, it is still suboptimal

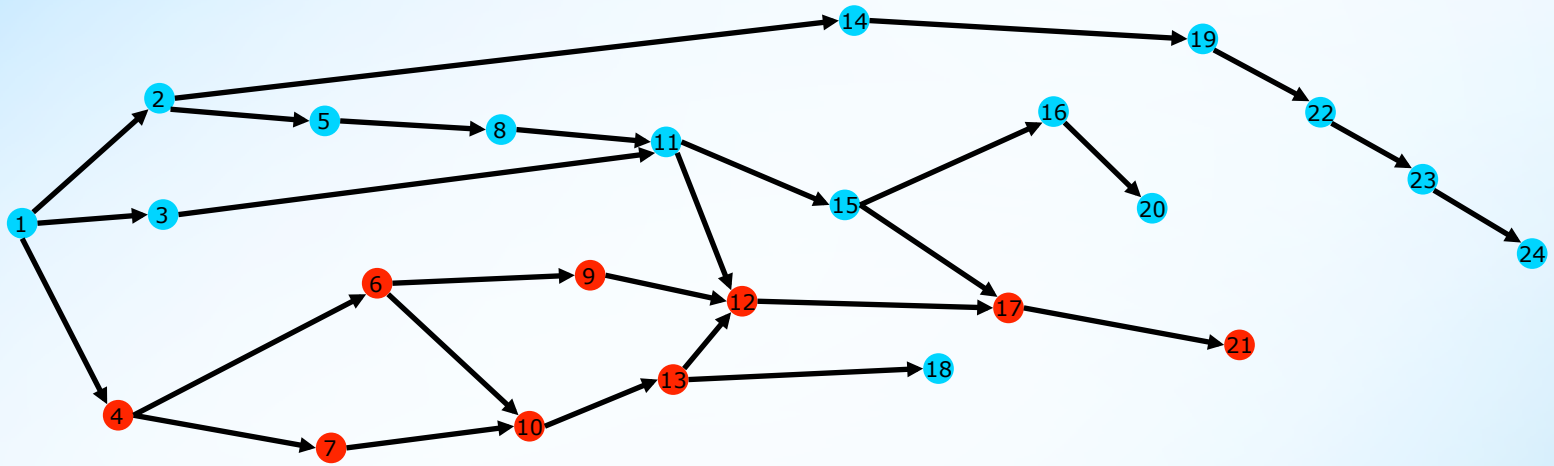
- CnC invokes the depends() method when a step instance is created (i.e. when a tag is put)
  - If all of the step's dependencies have been satisfied, the step will be scheduled
  - Otherwise, the step is queued up to wait on the data it will consume
- The problem is that depends() is called in *every frame* for each node that needs to be evaluated
  - However, the overall graph topology does not change from frame to frame
  - It is possible for different subgraphs to be evaluated from frame to frame, but eventually all possible subgraphs get cached
  - Thus, once a particular subgraph has been evaluated for the first time, the application is providing CnC with redundant dependency information in every subsequent frame that subgraph is evaluated
  - Parallelizing tag-puts speeds up the process, but does not change the fact that time is spent at the start of each frame doing redundant work

The ideal approach is to only generate dependency information once per subgraph



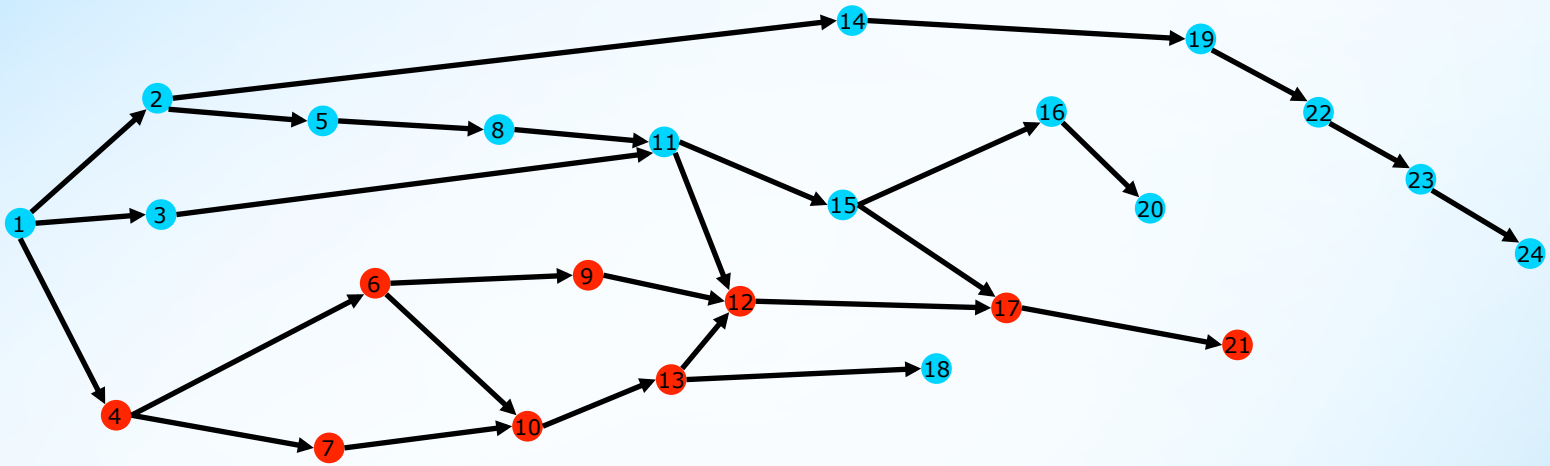
- We can cache dependencies for each subgraph at the application-level and then only put tags for nodes whose input dependencies have been satisfied
  - We maintain an atomic counter for each node to track whether all of its active predecessors have completed
  - For each node that completes evaluation, the counter for each of its successors will be decremented
  - When a node's counter reaches 0, a tag will be put for that node (no item collections are used)

For example, suppose we need to evaluate the subgraph containing the nodes in red



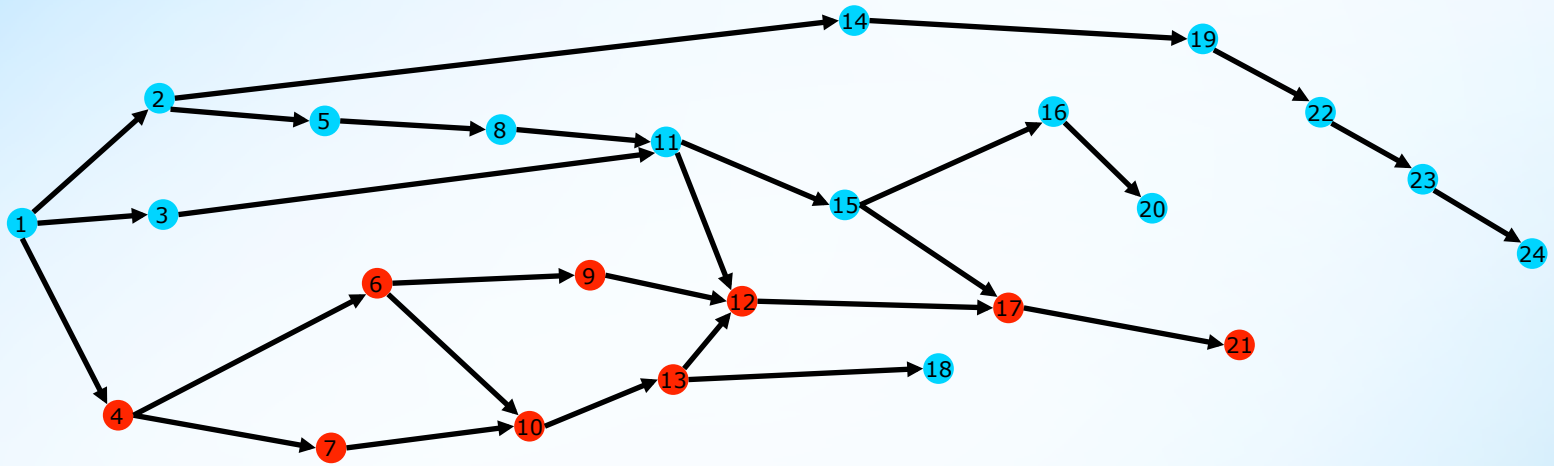
- If node 9 completes before node 13, then the step instance for node 13 will be the one that puts the tag for node 12
- If node 13 finishes before node 9, then node 9's instance will put the tag for node 12
- This approach avoids spending time on non-ready step instances, and is similar to that described in the IPDPS '10 paper by Chandramowlishwaran et al. (although the purpose of the technique in that paper was to avoid requeueing)

The problem with the delayed tag-put approach is that it moves further away from using "pure" CnC



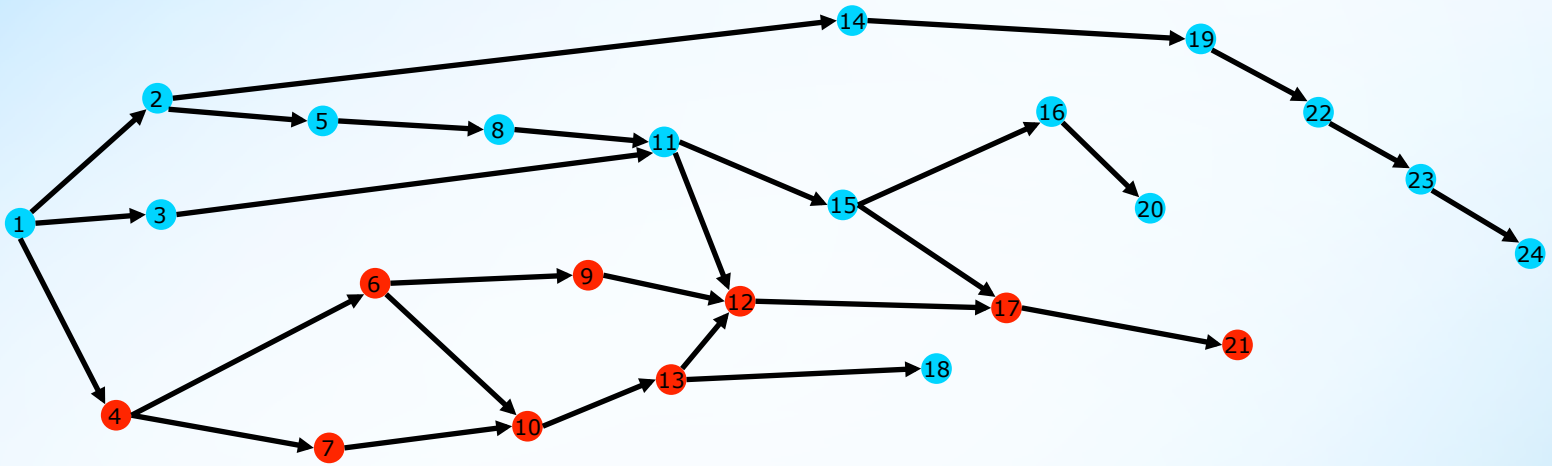
- By using dummy item collections, we were already avoiding the use of CnC for data management
- Avoiding the use of item collections and depends() means CnC doesn't have access to the dependencies in the program
  - This can make it more difficult to use other CnC tuning optimizations which rely on having dependency information
  - Now we are only using CnC as a higher-level scheduling engine on top of TBB
- I am currently looking at supporting this model inside of CnC, and some recent runtime system changes should help that effort

We can build on top of the delayed tag-puts optimization to "reuse" step instances



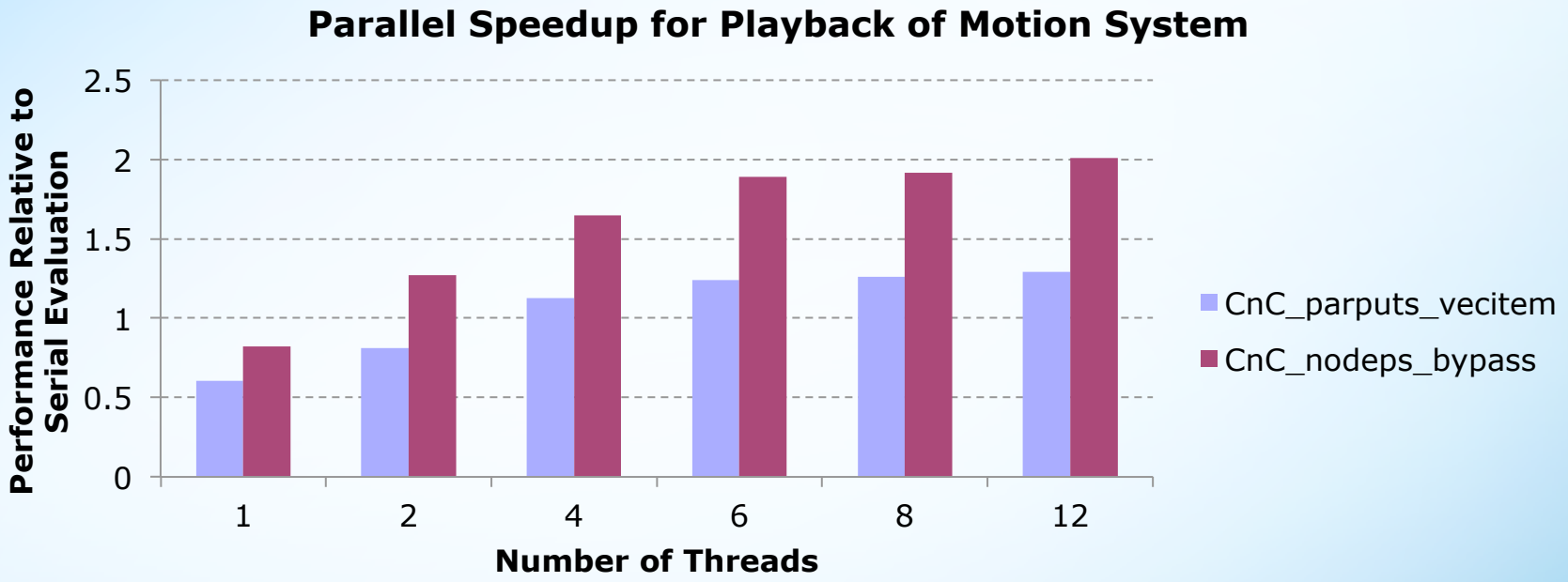
- For example, when node 4 completes, its step instance would normally put tags for nodes 6 and 7
- However, it's more efficient to only put a tag for node 7, and then loop around inside the step code to re-execute the step using node 6's tag information (possible because all steps have the same code)
  - This avoids the creation of a new CnC step instance (and TBB task) for node 6
  - It also has the potential to improve locality (although in practice the thread that executes node 4 would likely execute node 6 anyway)
- This optimization partially "bypasses" the CnC scheduler

As a result of this work, we actually did create a form of scheduler bypassing inside the CnC runtime system



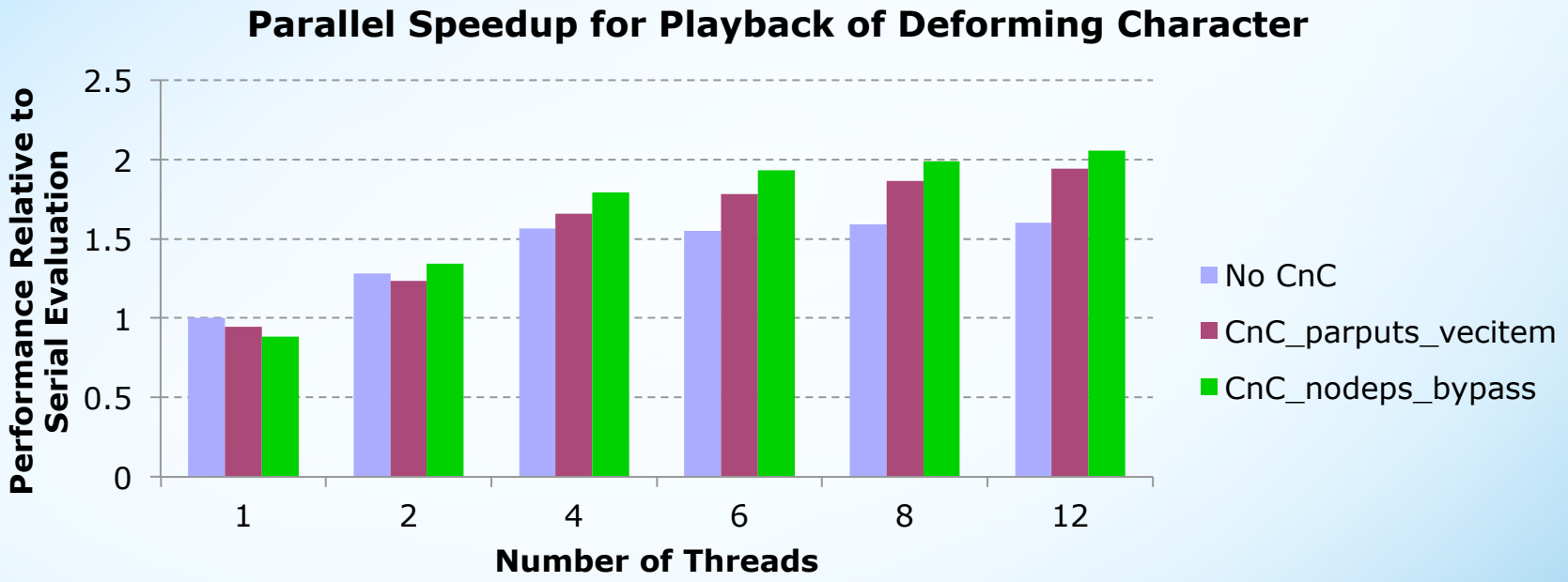
- However, it's not as efficient as our version of bypassing at the application level
- Inside the runtime system, CnC will enable a successor step instance to be executed within the same TBB task, bypassing TBB's scheduler
  - This is slightly more efficient than no bypassing, but still incurs the overhead of step instance creation for each node
  - There is the potential to improve on this approach inside CnC in conjunction with the runtime system version of delayed tag-puts that I mentioned earlier

Using delayed tag-puts and explicit bypassing improves CnC's performance



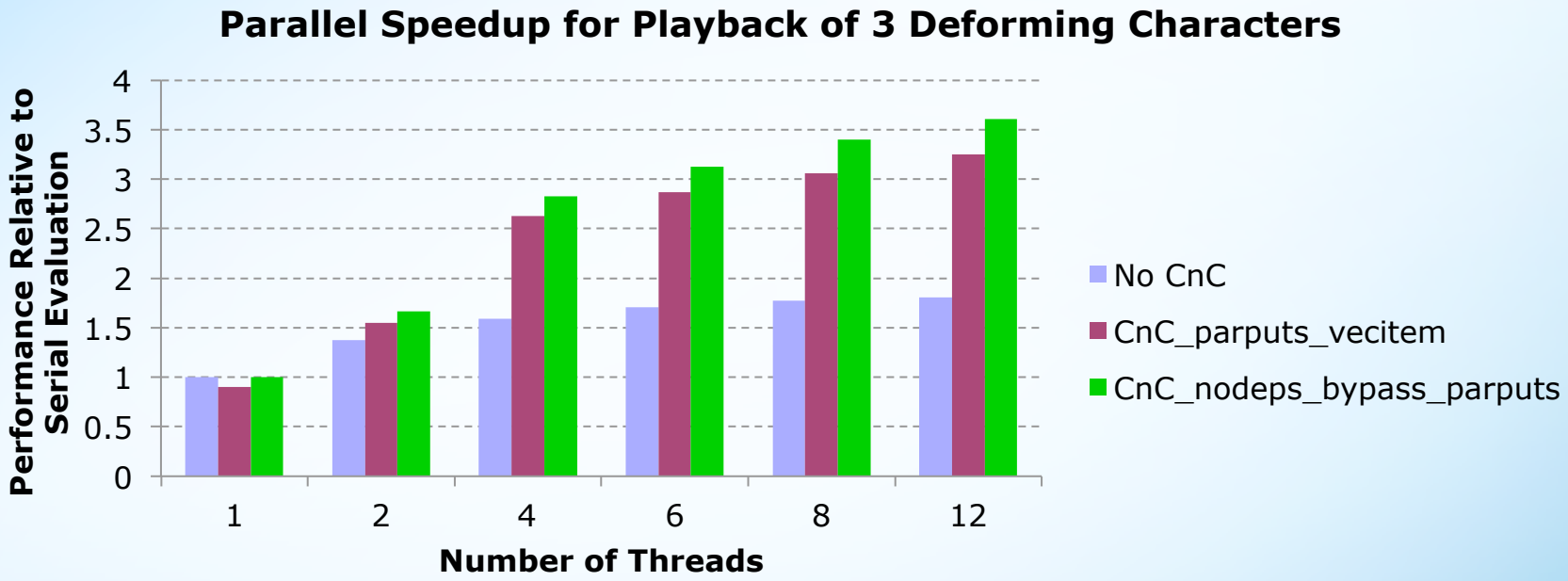
- The bar with delayed tag-puts ("nodeps") and scheduler bypassing does not parallelize the initial tag-puts, as the number of initially-ready nodes is small and parallelizing tag-puts slightly degrades performance at times
- Most of the performance benefit for the "nodeps_bypass" configuration comes from delaying the tag-puts
- The performance target for the motion system was a 2x speedup from graph-level parallelism, which is achieved with the full set of hardware threads

The performance gap between the two CnC approaches is not as noticeable for a deforming character



- With the more heavyweight deformation system included, CnC's runtime system overhead is not as prevalent as with just the motion system
- The narrower performance gap between the two CnC approaches makes it more feasible to consider switching back to the "purer" approach in order to facilitate other optimizations (switching back is a work in progress)

A multi-character workload provides more graph-level parallelism for CnC to exploit



- CnC provides a more significant advantage over only using internal node parallelism with 3 independent characters
- The most-optimized version ("nodeps_bypass_parputs") also parallelizes initial tag-puts from the environment since there are a greater number of ready nodes

Here is an outline of what we will be covering

- Performance results from a simple implementation
- Reduction of CnC runtime system overhead
  - Optimizations "inside" of the CnC model
  - Optimizations "outside" of the CnC model
- Additional optimizations to consider

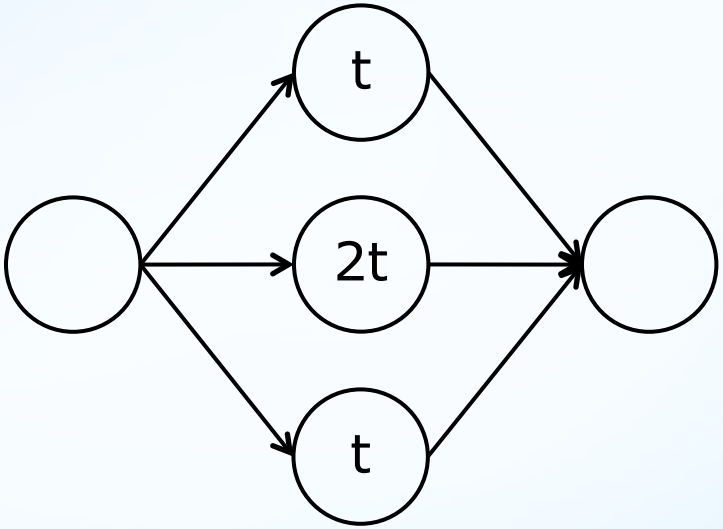
I've mainly focused in this talk on optimizations to reduce runtime system overhead

- That does not mean other areas of optimization were unexplored or unimportant
  - We have investigated and continue to explore additional optimizations
  - Up to this point, however, those experimental optimizations have not yet come to fruition for various reasons
- I will briefly discuss a few areas of optimization that are still being examined:
  - Using step priorities to guide scheduling (to target the critical path and improve locality)
  - Graph partitioning and the use of tag-ranges (to improve locality and reduce runtime system overhead)
  - Step affinity to processors (to improve locality)

CnC step priorities could in theory be used to guide scheduling, but…

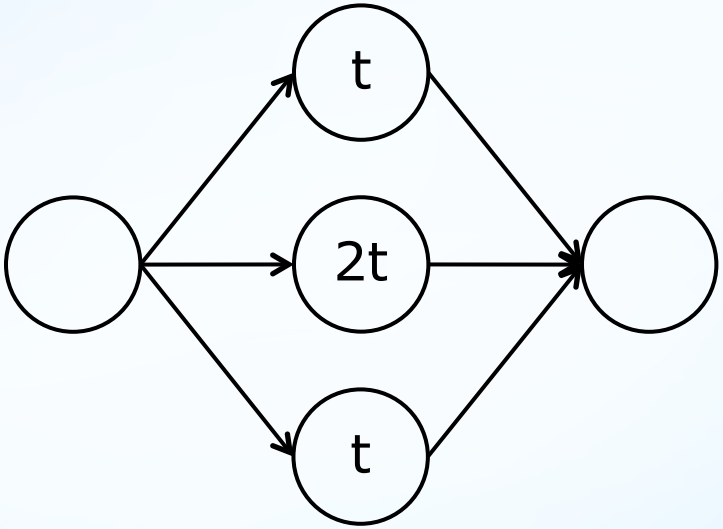
- …one item of note is that we haven't actually used priorities in the default scheduler for a couple of years
  - We had a scheduler implementation that supported step priorities in the 0.3 What If release of CnC
  - When we switched default schedulers, we dropped the use of priorities due to scalability concerns (so the step priority value was ignored)
- I spent a significant amount of time a while back trying to determine if it was worth using priorities for DWA workloads
  - I performed many experiments using a TBB approach of "faking" priorities, but didn't see much benefit
  - (The TBB "fake priorities" approach has since been deprecated, but coarse-grained task priorities are now supported)
  - I also collaborated with the TBB team to determine the potential for critical-path scheduling to improve performance for the DWA workloads; at the time, we came to the conclusion there wasn't much potential…

The TBB team has used the following example as a template to determine if priorities will provide a significant benefit



- On 2 threads, the worst-case scenario is that the middle nodes are executed in time *2t+t*; the best-case scenario is time *2t*
- The DWA deformation system (heaviest part of the graph) does not fit into the above model (heavyweight nodes essentially form a linear chain)
- The DWA motion system has the potential to fit into the above model, but nodes are so lightweight that there is little practical difference between runtimes of *t* and *2t*

However, there are still possible scenarios in which priorities could be beneficial



- For example, locality-aware scheduling could help to reduce memory pressure by prioritizing the consumers of a particular piece of data
- Another possibility is to use priorities at a coarse-grained level—e.g. a heterogeneous multi-character workload might have one "heavy" character and two "lighter" characters, which could fit into the model above
- I'm currently considering whether TBB's coarse-grained priorities would suffice, or whether finer-grained priorities are needed

Another optimization that we have explored is graph partitioning

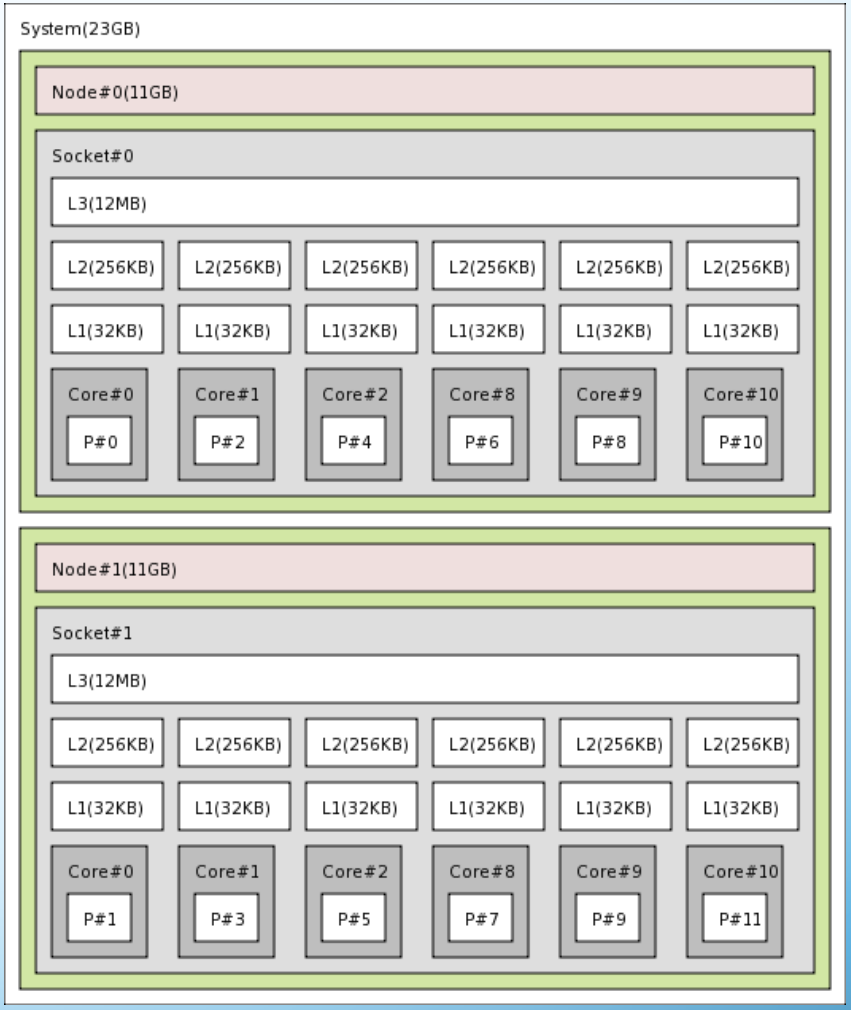
- We experimented with a general algorithm that relied on critical-path profiling
  - This approach generated good results
  - The problem was that it took a long time to run the algorithm, negating the benefit from partitioning
- As Martin mentioned, there are also some natural partitions to use (e.g. each of the limbs)
  - We did conduct some experiments in which we created per-limb partitions, and saw good performance improvements (e.g. >4x parallel speedup for the motion system)
  - One area of concern was the robustness of the approach—we relied on a node's name to determine the limb to which it belonged

We still plan to further investigate the potential for partitioning

- CnC's tag-ranges are a natural fit with our partitioning experiments
  - Tag-ranges are groups of tags that can be recursively split into smaller chunks
  - CnC can correctly handle intra-dependencies inside of a range and inter-dependencies between ranges
  - We plan to investigate the applicability of tag-ranges to the DWA project
- We had shelved our partitioning effort for a while to wait for a new model for the motion system
  - The new model in a sense performs a partitioning function for us, collapsing smaller nodes into larger ones
  - It's possible that with more heavyweight nodes in the motion system, CnC overhead will be less important

Optimizing a step's processor affinity has the potential to improve memory locality

- We experimented with pinning threads to cores
  - Mainly done for our scalability experiments
  - In general, our best manual assignment was about equivalent with the default Linux assignment
- We have also investigated mapping steps to the same software threads (without creating a partition)
  - However, there isn't a straightforward approach to do this in TBB
  - Creating partitions of steps is an alternative, although it reduces flexibility of execution



To sum up, multiple optimizations have been and continue to be explored

- CnC provides parallel speedup for DWA's workloads, but room for improvement remains
  - We have made progress in reducing CnC runtime system overhead, and continue to examine that issue
  - There are potential performance gains from optimizations such as different scheduling strategies
- Our collaboration with DWA has proven valuable in helping to expand CnC's capabilities for tuning
  - DWA has provided us with interesting workloads for experimenting with a variety of performance optimizations
  - Many of the tuning features we have added in the last couple of years have stemmed from the DWA project

Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2®, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Cilk, Core Inside, FlashFile, i960, InstantIP, Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

http://intel.com/software/products