Architektur und Programmierung von Grafik- und Koprozessoren

Rendering Algorithmen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2019

Paralleles Rendering

Paralleles Rendering

Verschiedene Ebenen:

- Parallelismus innerhalb von GPU.
- Paralleler Software Ray Tracer, parallel über alle Strahlen.
- Verteilte Speichersysteme.

Paralleles Rendering

Verteilter Speicher

- Wir interessieren uns für Paralleles Rendering mit verteiltem Speicher. Jeder Prozessor hat nur Speicher für einen Teil der Geometrie.
- Jeder Prozessor z. B. zuständig für Teil der Geometrie; Teil des zu rendernden Bildes; etc.
- Beispiel: Simulation auf Supercomputer, Simulationsergebnis liegt geteilt auf Cluster Knoten vor. Nun Post-Processing (Visualisierung) auf Cluster.

Ziel: Latenzvermeidung

Oberstes Gebot: reduziere Kommunikations-Overhead, da Netzwerkkommunikation teuer.

Abwägung: verteile Geometrie, verteile Teilbilder, etc.

Entscheidung von verschiedenen Parametern abhängig (wird z. B. Animation gerendert, gibt es evtl. zeitliche Kohärenz; Lastverteilung, “sparse” besetzte Daten ggf. anders zu behandeln als voll besetzte Datenstrukturen, etc.).

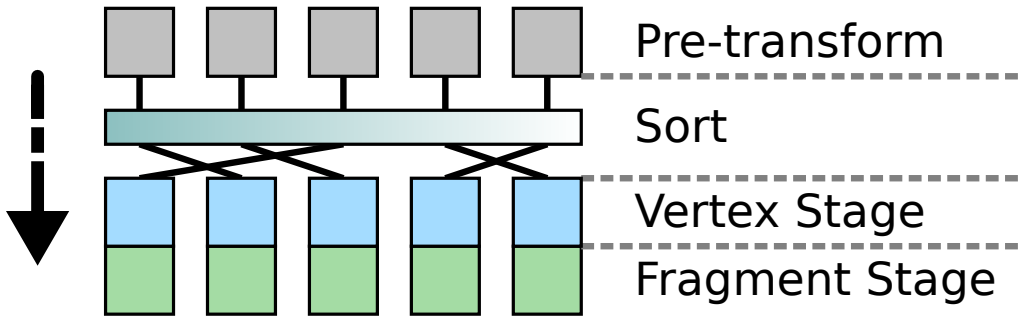
Molnars Taxonomie

Steve Molnar: A Sorting Classification of Parallel Rendering, Siggraph 1994.

- Paralleles Rendering als *Sortierproblem*:
  - Sortiere untransformierte oder teiltransformierte Geometrie.
  - Sortiere in Bildschirmkoordinaten transformierte Geometrie.
  - Sortiere Pixel oder Fragmente.
- Unterscheide danach, *wann* sortiert wird.
- Taxonomie orientiert sich an Algorithmus RASTERISIERUNG. Spielt bei GPU Hardware Design große Rolle.
- Auch auf Software Rendering anwendbar (z. B. GPU Cluster, Software sortiert und verteilt Daten per Netzwerk).

Molnars Taxonomie

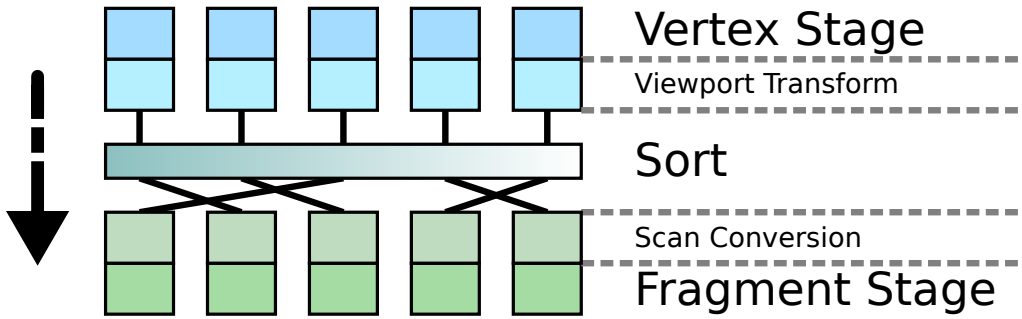
Sort-First



- Verteile im wesentlichen untransformierte *Primitive* auf Prozessoren.
  - Anfangs Verteilung beliebig.
  - *Pre-Transform*: transformiere Primitive gerade so weit, um zu wissen, welcher Fensterbereich überlappt (ggf. wird Transformation nicht einmal angewendet).
  - Umverteilung (Sort) an Prozessoren, die für Fensterbereich zuständig.
  - Frame-to-Frame Kohärenz $\Rightarrow$ ggf. nur anfangs hohe Last auf Interconnect.

Molnars Taxonomie

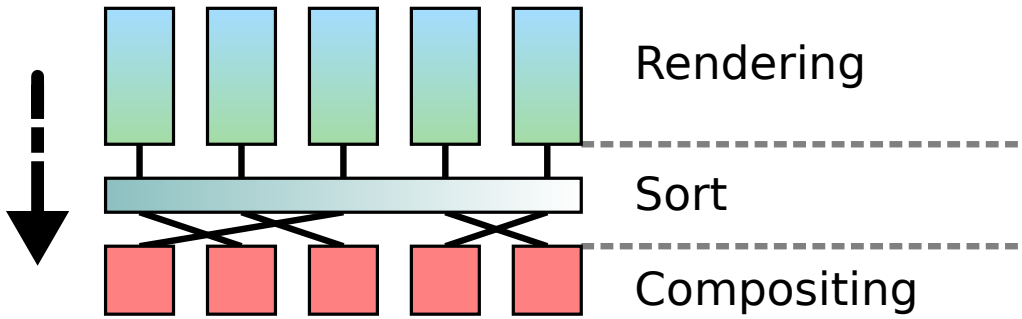
Sort-Middle



- Verteile *Primitive in Bildschirmkoordinaten* auf Prozessoren (Raster Engines).
  - Gesamte Geometrie Phase und Primitive Assembly durchlaufen.
  - Sort vor Scan Conversion.
  - Sort-Middle typisch für Hardware Rendering, kaum auf Software Rendering anwendbar.

Molnars Taxonomie

Sort-Last



- Verteile vollständig gerenderte Fragmente oder Pixel auf Prozessoren.
  - Erst vollständige Rendering Phase (egal welcher Algorithmus).
  - Sende *Intermediärbilder* über Interconnect an zuständige Prozessoren (Sort).
  - Compositing auf einem oder mehreren Prozessoren.

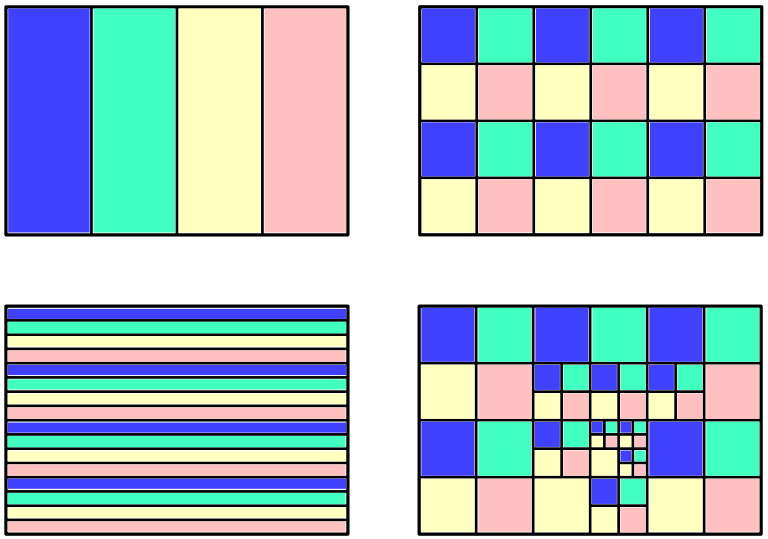
Molnars Taxonomie

Compositing

- Prozessoren für Teile der Geometrie zuständig - verwalte z. B. mittels k-d tree.
- Mit Hilfe des k-d trees können die Intermediärbilder tiefensortiert werden.
- Teiltransparente Geometrie: *Alpha Compositing* - Intermediärbilder mit Alpha Kanal, Compositing mit Over Operation (vgl. Alpha Blending).
- Opake Geometrie: *Depth Compositing*: Intermediärbilder mit Tiefenkanal, setze auf Basis von Tiefeninformation zusammen.

Molnars Taxonomie

Lastverteilung



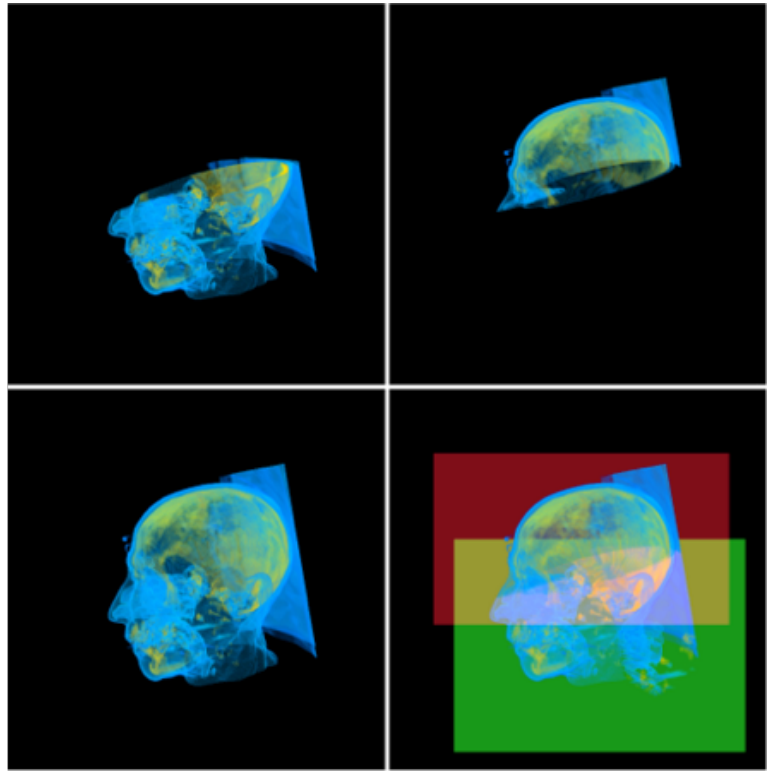


Abbildung: Links: Sort-First statische und dynamische Lastverteilung, rechts: Sort-Last mit zwei Prozessoren ohne Lastverteilung.

Molnars Taxonomie

Dynamische Lastverteilung

Bei dynamischer Lastverteilung und z. B. Sort-First wird das Bild a priori in Bildausschnitte unterteilt, die Zuweisung von Bildausschnitten zu Prozessoren geschieht jedoch dynamisch.

Bildausschnitte (z. B. Kacheln) können dazu etwa in eine Schlange einsortiert werden, Prozessoren holen Kachel aus Schlange, sobald Arbeit verrichtet.

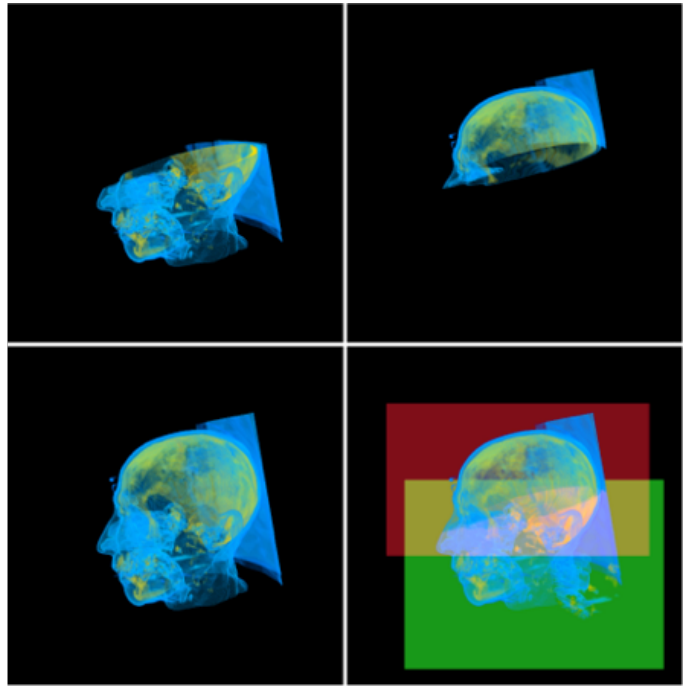
Kann mit adaptiven Verfeinerungsverfahren gepaart werden (arbeitsintensivere Regionen werden dynamisch in kleinere Kacheln unterteilt).

Bemerkung: komplizierte Lastverteilungsalgorithmen können ungünstiges Branching Verhalten etc. mich sich bringen.

Sort-Last Compositing Algorithmen

- Bei vielen Prozessoren ist nicht Rendering das Bottleneck, sondern Compositing.
- Problem ist weniger die Komplexität des Verfahrens, sondern der hohe Bandbreitebedarf.
- Beim naiven Verfahren: Bandbreitebedarf verteilt sich ungleichmäßig auf Interconnect, Verbindung zum Anzeigeknoten überlastet.

Sort-Last Compositing Algorithmen

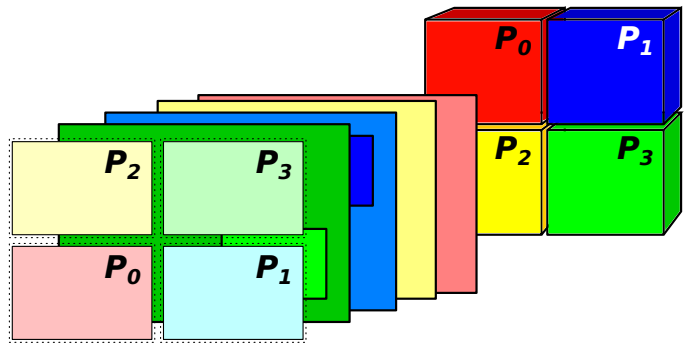


I. Allg. werden beim Sort-Last Verfahren in der letzten Phase vollaufgelöste Bilder zwischen Prozessoren hin- und her geschickt.

Naiv: jeder Prozessor rendert vollaufgelöstes Bild und schickt es zum Compositing an *Anzeigeknoten*. Schlechtes Skalierungsverhalten schon bei moderat vielen Prozessoren.

Sort-Last Compositing Algorithmen

Direct-Send



Jeder von P Prozessoren für $\frac{1}{P}$ Teil des Bilds verantwortlich. Alle Prozessoren rendern, senden danach $(P-1) \times \frac{1}{P}$ Bildausschnitte an die P anderen Prozessoren. Jeder Prozessor führt Compositing für seinen Bildteil durch. Anzeigeknoten sammelt Bilder ein.

Sort-Last Compositing Algorithmen

Direct-Send

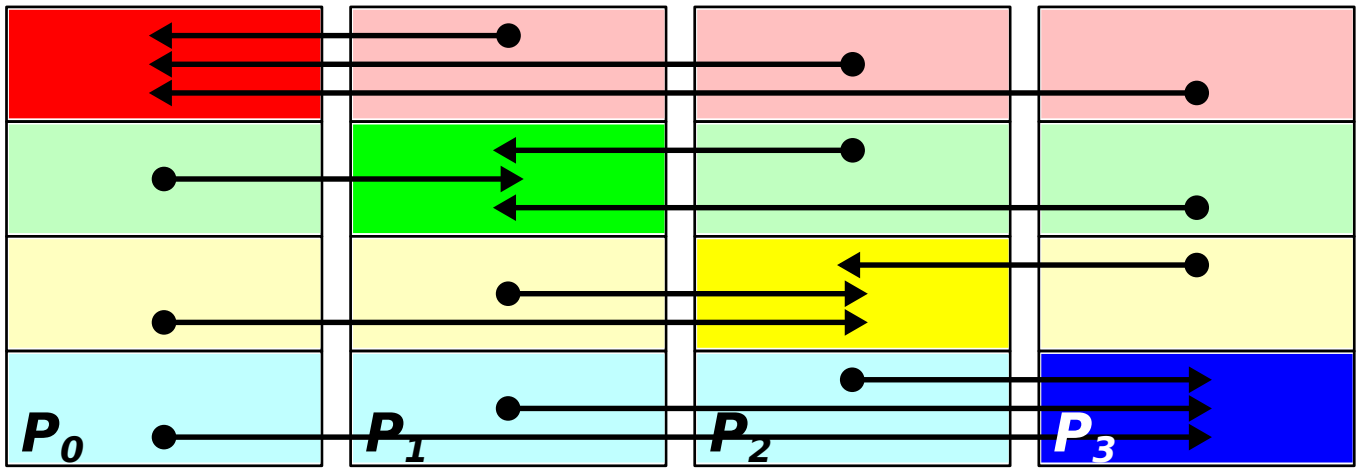


Abbildung: Kommunikationsmuster beim Direct-Send Compositing, vgl. Bethel et al.: High Performance Visualization - Enabling Extreme Scale Scientific Insight (2013).

Sort-Last Compositing Algorithmen

Baumbasiertes / Rundenbasiertes Compositing

- Teile Compositing in *Runden* auf, jede Runde entspricht einem Level in Baum.
- Direct-Send: Baum der Höhe 1, alle Arbeit innerhalb einer Runde.
- Ziel: bessere Balance zwischen Anzahl gleichzeitiger Nachrichten sowie Bandbreitenbedarf einzelner Nachrichten.

Sort-Last Compositing Algorithmen

Baumbasiertes / Rundenbasiertes Compositing

- Naive Implementierung: schicke in jeder Runde Intermediärbild an direkten Nachbarn. Dieser führt Compositing durch. In der nächsten Runde ist aus dem Prozessorpaar nur noch der Nachbar aktiv.
- Am Ende liegt das zusammengesetzte Bild im Speicher *eines* Prozessors.

Sort-Last Compositing Algorithmen

Baumbasiertes / Rundenbasiertes Compositing

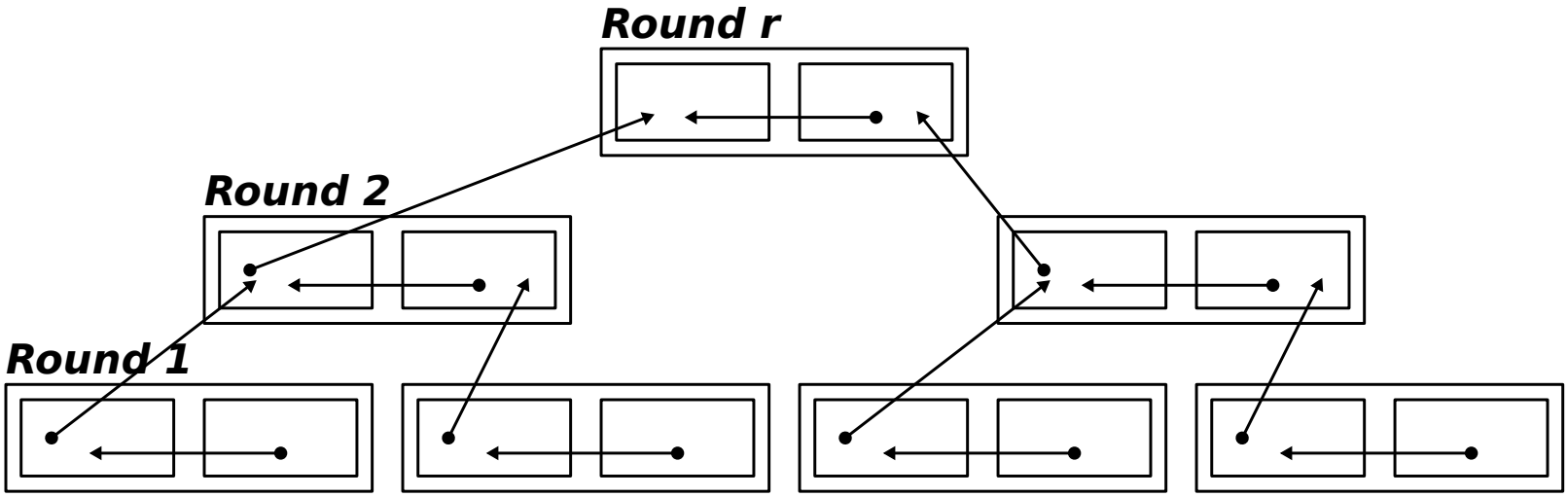


Abbildung: Problem bei naiv implementiertem rundenbasiertem Compositing (und generell bei binärbaumbasierten parallelen Algorithmen wie Reduce): Viele Prozessoren, die in frühen Runden arbeiten, sind in späteren Runden nicht beschäftigt.

Sort-Last Compositing Algorithmen

Binary-Swap

Binary-Swap Compositing löst dieses Problem,

- indem pro Runde zwei Prozessoren paarweise für eine Hälfte ihres *gemeinsamen* Bildbereichs zuständig sind (*binary*),
- und indem diese die Bilddaten, für die jeweils der andere Prozessor zuständig ist, in jeder Runde tauschen (*swap*).

In jeder Runde wird zudem der Bildausschnitt, für den zwei Prozessoren zuständig sind, halbiert. Außerdem wird die Distanz zwischen den Prozessorpaaren verdoppelt ("distance-doubling and vector-halving" Algorithmus).

Es sind stets P Prozessoren beschäftigt, wobei $P = 2^k$.

Ist der Binary-Swap Algorithmus durchgelaufen, liegt das zusammengesetzte Bild *verteilt* auf P Prozessoren vor.

Sort-Last Compositing Algorithmen

Binary-Swap

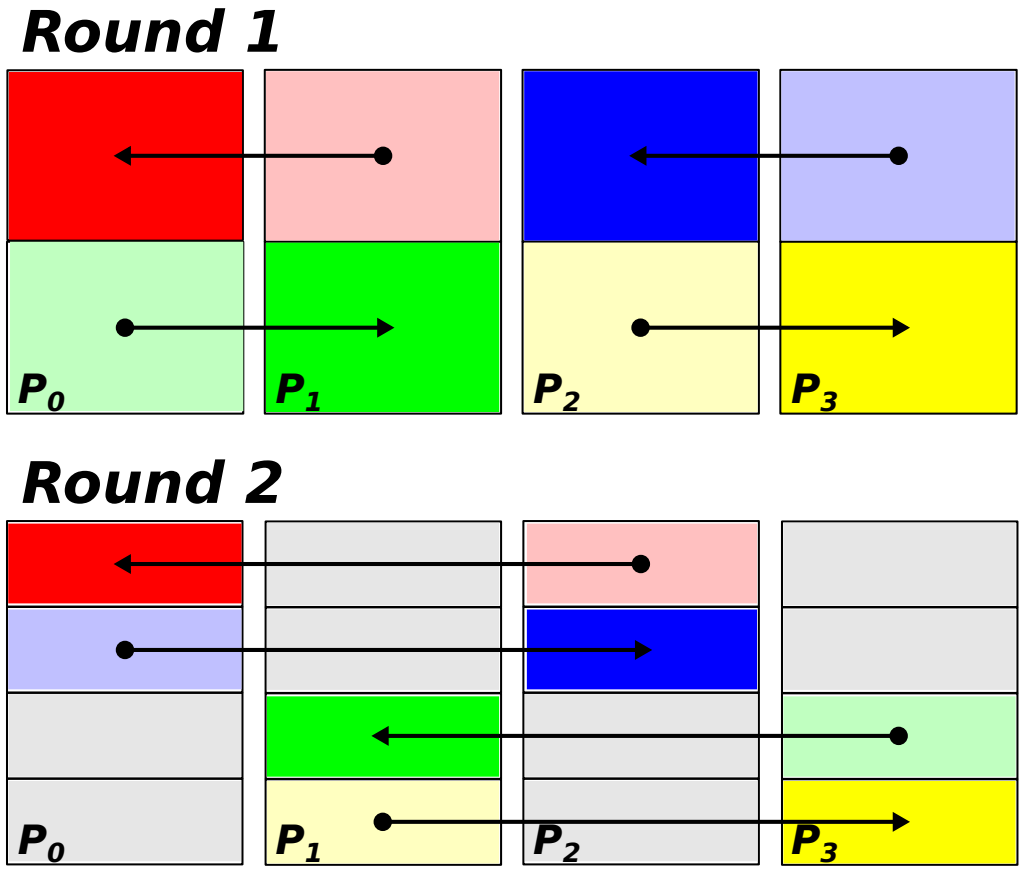


Abbildung: Kommunikationsmuster beim Binary-Swap Compositing mit vier Prozessoren, vgl. Bethel et al.: High Performance Visualization - Enabling Extreme Scale Scientific Insight (2013).

Sort-Last Compositing Algorithmen

Binary-Swap

- Bemerkung: in jeder Runde führen Paare von Prozessoren Direct-Send durch.
- Bemerkung: der Binary-Swap Algorithmus skaliert nur für $P = 2^k$ Prozessoren.

Sort-Last Compositing Algorithmen

Verallgemeinerung

Wir wollen die Algorithmen *Direct-Send* und *Binary-Swap* verallgemeinern. Dazu führen wir die folgende Notation ein.

Bezeichne r die Anzahl Runden, die der Compositing Algorithmus durchführt. Bezeichne k_i die Anzahl an Prozessoren, die in jeder Runde in jeder Kommunikationsgruppe beteiligt ist. Es ergibt sich der "k-Vektor" $\vec{k} = [k_1, k_2, ..., k_r]$.

Sort-Last Compositing Algorithmen

Verallgemeinerung

Sei P die Anzahl aller Prozessoren. Dann ergibt sich z. B.

Direct-Send:
$r = 1,\ \vec{k} = [P]$.

Rundenbasiert (naiv & Binary-Swap):
$r = log(P),\ \vec{k} = [2, 2, 2, ...]$.

Sort-Last Compositing Algorithmen

Radix-k

Frage: sind andere Kombinationen von r und $\vec{k}$ möglich?

Antwort: ja. Der *Radix-k* Compositing Algorithmus erlaubt jede Faktorisierung von P, sodass

$$\prod_{i=1}^{r} k_i = P, \tag{38}$$

wobei in jeder Runde i alle Kommunikationsgruppen die gleiche Größe k_i haben. In jeder Kommunikationsgruppe selbst wird *Direct-Send* durchgeführt.

Sort-Last Compositing Algorithmen

Radix-k

Sei etwa $P = 12$. Mögliche gültige Konfigurationen für den Radix-k Algorithmus sind z. B.:

$r = 1 : \vec{k} = [12]$
$r = 2 : \vec{k} = [6, 2], \vec{k} = [2, 6], \vec{k} = [3, 4], \vec{k} = [4, 3]$
$r = 3 : \vec{k} = [2, 2, 3], ...$
$r = ...$

Sort-Last Compositing Algorithmen

Radix-k

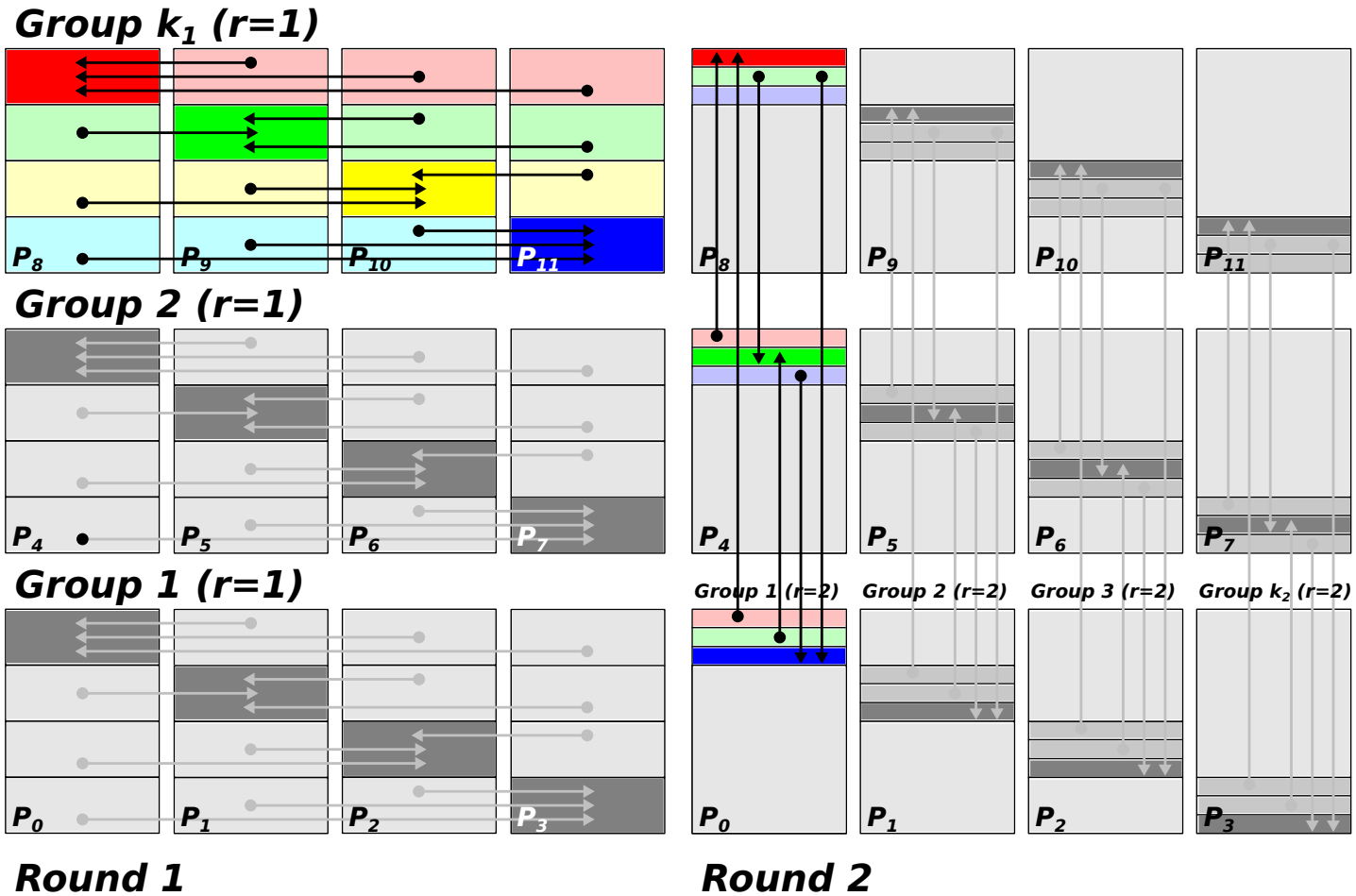


Abbildung: $P = 12, r = 2, \vec{k} = [4, 3]$, vgl. Bethel et al.: High Performance Visualization - Enabling Extreme Scale Scientific Insight (2013).

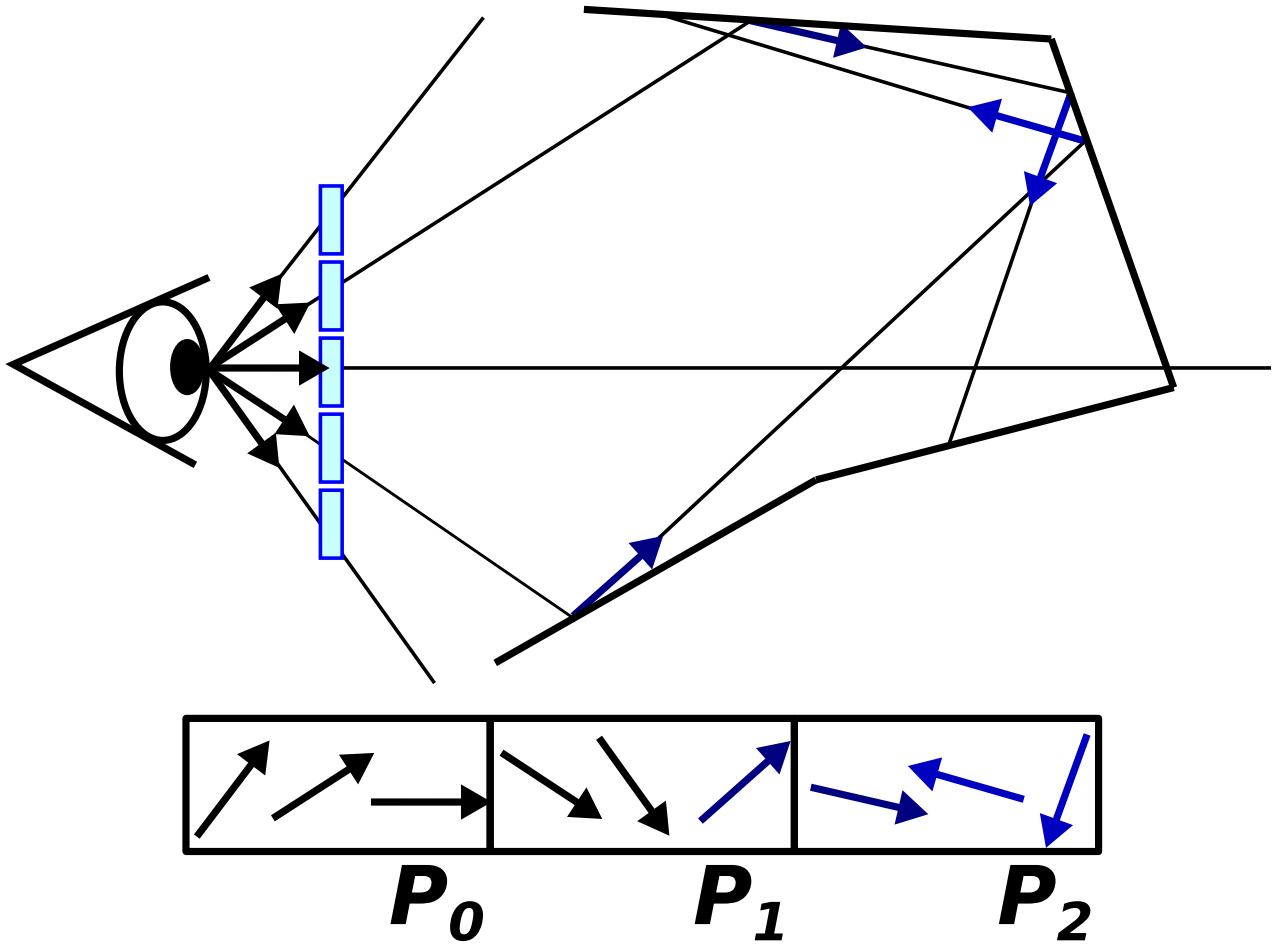
Sort-Last Compositing Algorithmen

Radix-k

Bemerkung: sowohl Direct-Send als auch Binary-Swap sind gültige Radix-k Konfigurationen (naiv rundenbasiertes Compositing nicht, da innerhalb der Kommunikationsgruppen kein Direct-Send).

Andere Parallelisierungsarten

Strahlverfolgung mit Wavefronts



Andere Parallelisierungsarten

Strahlverfolgung mit Wavefronts

- Halte mit jedem Bounce die gesamte Pipeline an.
- Verteile alle Strahlen für nächsten Bounce auf P Prozessoren.
- Vorher: sortiere ggf., z. B. nach Material-ID oder so, dass Strahlen mit ähnlichem Ursprung und ähnlicher Richtung nah beieinander.
- Kombiniere mit “compaction”: schiebe inaktive Strahlen “nach rechts” in der Liste, prozessiere nur den aktiven Teil der Liste, z. B. mit SIMD.
- Wavefronts dann besonders nützlich, wenn Strahlen sehr inkohäherent.

Andere Parallelisierungsarten

Distributed Shared Memory

- Verteiltes Speichersystem.
- Software Layer, das z. B. Cache Kohärenz Protokoll implementiert.
- Nachrichtenversand auf Basis von MPI.
- z. B. anwendbar bei Sort-First: Geometrie verteilt sich auf Speicher, dieser wirkt aber, als wäre er geteilt.

Recap (1)

- Lichttransportgleichung, Computergrafik Grundlagen: Scan Konvertierung, Texturierung, Tiefenpuffer, Alpha Blending.
- Grafik Pipeline mit Rasterisierung, Algorithmus $O(V \times VP \times L)$.
  - Einteilbar in mehrere parallele *Stages*:
  - Vertex Stage.
  - Rasterisierung.
  - Fragment Stage.
- Deferred Shading für viele Lichter: $O(V \times VP + L \times VP)$.
  - GPU Implementierung: speichere g-Buffer in fensterfüllenden Texturen.
  - Erhöhter Speicher- und Bandbreitenbedarf.

Recap (2)

- Strahlverfolgung: mittlerweile Hardware Unterstützung auf manchen GPUs (DXR & RTX), i. Allg. auf GPUs parallel implementierbar.
- Flexibler: mehr Strahlen für höheren Realismus. Suchdatenstrukturen: Komplexität $O(log(V))$.
- Paralleles Rendering als Vorbereitung auf GPU Architekturen.

Literaturempfehlungen

- Peter Shirley, Steve Marschner: Fundamentals of Computer Graphics, 3rd ed. (2009)
- Matt Pharr, Wenzel Jakob, Greg Humphreys: Physically Based Rendering - From Theory to Implementation, 3rd ed. (2017)
- E. Wes Bethel, Hank Childs, Charles Hansen: High Performance Visualization - Enabling Extreme-Scale Scientific Insight, 1st ed. (2013)
- Juan Pineda: A Parallel Algorithm for Polygon Rasterization, Siggraph (1988)
- Turner Whitted: An Improved Illumination Model for Shaded Display, Communications of ACM (June 1980)